MASTER EN SCIENCES INFORMATIQUES

Définition et implémentation d'un langage déclaratif pour l'analyse d'audit trails

Gérard, Françoise

Award date:
1998

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX (FUNDP)
NAMUR
INSTITUT D'INFORMATIQUE
RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

Définition et implémentation d'un langage déclaratif pour l'analyse d'audit trails

Françoise GÉRARD

Mémoire présenté pour l'obtention du grade
de Licencié en Informatique

Année académique 1997 - 1998

Remerciements

Je tiens à remercier tout particulièrement mon promoteur, le Professeur Baudouin Le Charlier, pour son implication dans ce mémoire et son active collaboration lors du développement du langage LaDAA qui, sans lui, n'aurait sans doute jamais vu le jour. Merci à lui de m'avoir fait profiter de ses remarques pertinentes et indispensables.

Mes plus vifs remerciements s'adressent aussi au Docteur Abdelaziz Mounji pour le temps et l'énergie qu'il a consacrés à me faire partager ses connaissances d'ASAX, pour ses encouragements, ses nombreux conseils et sa patience à mon égard.

Merci aussi au Docteur Naji Habra et à Messieurs Radu Cotet et Jean Henrard pour leur disponibilité et les différents renseignements et documents qu'ils m'ont fournis.

Je remercie également Joachim, Ghislain, Jean-Marie, Majid, Xavier, Olivier, Vincent, Virginie, Caroline, Patrick, Jean-François, Jean-Pol, Cédric, Sébastien, Efrem... toutes ces personnes qui, de près ou de loin, ont contribué au bon déroulement de cette année académique et de ce mémoire, et m'ont fait passer des moments inoubliables.

Enfin, je tiens à adresser un merci tout particulier à mes parents qui voient ici s'achever un bien long cheminement fait de patience, d'encouragements et de soutien, et sans qui ces études n'auraient jamais pu aboutir.

Table des matières

Table des figures

Résumé

L'objet de ce mémoire consiste en la définition d'un nouveau langage de programmation de style déclaratif spécialement conçu pour extraire d'une séquence quelconque donnée une sous-séquence particulière vérifiant une condition précise. Cette condition peut être extrêmement simple ou plus élaborée. Nous verrons en outre comment ce langage peut être utilisé dans le problème de l'analyse efficace de fichiers d'audit au travers de l'outil ASAX.

Abstract

The purpose of this thesis is to define a new declarative language especially designed to extract, from a given sequence, a particular subsequence matching some more or less elaborated conditions. We will also see how this language can be used for efficient audit trail analysis by the way of ASAX.

Introduction

Ce mémoire se situe dans le cadre du projet ASAX (Advanced Security Audit-trail analysis on Unix) et a pour objectif de définir et implémenter un nouveau langage de programmation plus déclaratif que le langage actuellement utilisé dans ASAX, à savoir le langage RUSSEL (RUle-baSed Sequence Evaluation Language).

Le nouveau langage, que nous avons baptisé LaDAA (Langage Déclaratif d'Analyse d'Audit trails), a donc pour but de permettre à l'utilisateur de spécifier de façon tout à fait déclarative le scénario d'attaque recherché. Le langage LaDAA est en partie basé sur le concept des expressions régulières, concept qui est quelque peu étoffé et modifié dans le cadre de ce langage. Ainsi, l'utilisateur du langage LaDAA n'aura plus à se soucier d'écrire des règles qui se déclenchent entre elles en fonction du fichier analysé (comme c'est le cas en RUSSEL), mais pourra exprimer le scénario recherché sous forme d'une requête déclarative.

En outre, les automates finis non déterministes jouent un rôle clé dans l'implémentation du langage LaDAA. En effet, nous verrons que la traduction directe du langage LaDAA en langage RUSSEL ne s'impose pas de façon aisée. Par contre, la transformation des automates en code RUSSEL peut se faire de façon tout à fait immédiate, comme cela a été exposé dans [2].

Ce mémoire sera divisé en cinq chapitres. Dans le premier, nous faisons quelques rappels sur ASAX, les fichiers d'audit trails et le langage RUSSEL. Nous y expliquons également les raisons d'être du nouveau langage et voyons la relation entre celui-ci et les expressions régulières. Le deuxième chapitre introduit de manière intuitive le langage LaDAA. Les chapitres trois et quatre sont respectivement consacrés à l'exposé de la syntaxe et de la sémantique du langage LaDAA. Le cinquième chapitre traite, pour sa part, de l'implémentation du langage, à savoir la traduction de celui-ci en langage RUSSEL via l'utilisation des automates finis non déterministes. Pour conclure, nous donnons des extensions possibles du nouveau langage.

Chapitre 1

Entrée en matière

Nous présentons brièvement ici le projet ASAX, ainsi que le langage RUSSEL. Nous expliquons également en quelques mots ce qu'est un fichier d'audit trail. Cette partie peut être allègrement survolée par le lecteur familiarisé avec ASAX. Nous exposons ensuite les raisons d'être d'un nouveau langage pour ASAX. Finalement, nous montrons en quoi le langage LaDAA est inspiré du concept des expressions régulières et surtout en quoi ce concept est modifié et étoffé pour le langage.

1.1 Projet ASAX

1.1.1 Historique du projet

Historiquement, ASAX a été développé en tant que système d'analyse d'audit trails pour BS2000 et SINIX, deux systèmes d'exploitation radicalement différents de Siemens-Nixdorf A.G. Il faut souligner que la conception et l'implémentation d'ASAX ont été menées en parallèle avec l'établissement du système d'audit du BS2000. Ainsi, au moment même où ASAX était en cours de développement, peu de connaissances étaient disponibles à propos des informations contenues dans les enregistrements des fichiers d'audit, ni même à propos de leur format.

Sachant cela, la conception d'ASAX a été conduite selon deux buts majeurs. Premièrement, la conception d'un format d'enregistrement canonique hautement flexible, dans lequel des enregistrements de systèmes d'audit spécifiques peuvent être traduits de façon tout à fait directe. Deuxièmement, la conception d'un langage efficace et puissant conçu pour exprimer et détecter des scénarios complexes d'audit.

1.1.2 Fichiers d'audit normalisés

Généralement, les systèmes d'exploitation génèrent une description de certains des aspects bien choisis de leur comportement. Un fichier généré de cette façon est appellé *audit trail* ou *fichier d'audit.*

Un format de fichier d'audit, le format NADF (Normalized AuDit file Format), a été défini pour les besoins du projet ASAX. Ce format a été conçu extrêmement simple et flexible, de sorte que tout fichier d'audit peut être

traduit sans problème en fichier sous format NADF. Des *adaptateurs de format* spécifiques peuvent être écrits pour réaliser la traduction de fichiers natifs[1] particuliers en fichiers normalisés. L'expérience faite avec BS2000 et SINIX montre que ceci peut être réalisé de façon immédiate.

Un fichier sous format NADF (appelé aussi fichier NADF) est composé d'une *séquence chronologique d'enregistrements.* C'est donc un fichier séquentiel d'enregistrements. Chaque enregistrement est lui même constitué de sa propre longueur et d'une liste de données d'audit, chacune représentée par un identifiant, la longeur de la donnée en question et la valeur prise par la donnée. Le type de la donnée est uniquement déterminé par son identifiant. Remarquons que le format d'enregistrement est identique pour chacun des enregistrements du fichier NADF. Toutefois, certaines données peuvent ne pas prendre de valeur pour un enregistrement particulier dans le fichier, et être dès lors non définies pour cet enregistrement.

1.1.3 Objectif d'ASAX

L'ultime objectif du projet ASAX est de définir et implémenter un système universel, efficace et puissant pour l'analyse d'audit trails.

Universalité : l'architecture

Pour être universel, l'outil développé doit pouvoir tourner sur une large gamme de systèmes d'exploitation. Pour cela, il doit être possible d'analyser tout fichier d'audit trail, pourvu que celui-ci soit précédemment traduit dans un format approprié. C'est dans ce but que le format NADF dont nous avons parlé plus haut a été défini. Ce format a été conçu suffisamment flexible pour que tous les fichiers d'audit existants puissent être traduits dans celui-ci de façon directe. Ainsi, la simplicité du format NADF garantit la portabilité au niveau physique. Une fois les fichiers d'audit traduits en format NADF, l'analyse sera faite exclusivement sur les fichiers normalisés.

Remarquons toutefois qu'une variante d'ASAX permet d'effectuer (dans ce qui est appelé le *mode évaluation/conversion*) l'analyse de fichiers natifs. Nous n'entrerons cependant pas ici dans ce genre de détails et conseillons au lecteur curieux de se référer à [2] pour en savoir plus sur le sujet. Nous supposons dès lors que toute analyse est effectuée en *mode standard* sur des fichiers normalisés.

Puissance : le langage

RUSSEL est un langage à base de règles conçu spécialement pour l'analyse d'audit trail. L'idée est de fournir un langage ayant la puissance généralement reconnue des langages à base de règles "classiques" (voir [1] et ses références), mais spécialement prévu pour le traitement efficace de grands fichiers séquentiels.

Toutefois, dans le cas de l'analyse de fichiers d'audit, un langage à base de règles ne peut permettre d'encoder n'importe quel genre de connaissance. Dans

[1] Nous appelons *natif* tout fichier d'audit sous un autre format que le format NADF.

cette optique, un tel langage sera utilisé d'une façon précise: reconnaître des patterns particuliers dans un fichier séquentiel et déclencher les actions appropriées, comme par exemple l'activation d'une alarme ou l'envoi d'un message. L'idée de RUSSEL est de profiter de cette utilisation particulière pour rendre le langage aussi efficace et facile d'utilisation que possible.

Ainsi, RUSSEL peut être vu comme un langage procédural incorporant une structure de contrôle prédéfinie particulièrement appropriée pour raisonner sur des séquences. Sa structure générale est basée sur un mécanisme de déclenchement de règles qui peut être décrit comme suit:

- Le fichier d'audit est traité séquentiellement enregistrement après enregistrement par un mécanisme de déclenchement de règles (le fichier est ainsi parcouru une seule fois). A un moment donné, il y a un enregistrement courant en cours d'analyse et une collection de règles qui sont actives.
- Les règles actives englobent toutes les connaissances pertinentes à propos du passé de l'analyse en cours. Cette connaissance est dès lors appliquée à l'enregistrement courant en exécutant les règles actives pour celui-ci. Ce procédé génère à son tour de nouvelles règles qui seront appliquées plus tard.
- Du point de vue du programmeur, une règle ressemble à une procédure paramétrée classique qui peut contenir un type d'action particulier: le déclenchement de règle.
- Déclencher une règle implique d'en fournir les paramètres effectifs ainsi que le moment où la règle doit être déclenchée: pour l'enregistrement courant, le suivant ou à la fin de l'analyse.
- Le procédé est initialisé par un ensemble de règles activées pour le premier enregistrement.

Dès lors, programmer en RUSSEL une requête pour l'analyse de fichiers d'audit consiste en l'écriture d'un certain nombre de règles adéquates qui seront activées selon le schéma global exposé ci-dessus. Il apparaît donc clairement que le fait que les fichiers d'audit soient présentés de manière chronologique joue un rôle essentiel dans cette approche, l'analyse avancée d'audit trails consistant principalement en la recherche de séquences chronologiques complexes dans le flux de tous les événements enregistrés.

Efficacité: l'implémentation

L'efficacité est un aspect critique de l'analyse d'audit trails en raison de la quantité de données qui doivent être analysées. L'efficacité nécessaire est atteinte grâce à deux principes clés de la conception d'ASAX:

1. l'analyse de l'audit trail est faite en une et une seule passe et
2. les étapes répétitives sont optimisées autant que possible (traitement de l'enregistrement courant par un ensemble de règles actives et passage d'un enregistrement au suivant).

Le premier principe est réalisé par le langage RUSSEL qui est précisément conçu pour permettre à n'importe quelle requête d'être traitée en une seule passe. L'information nécessaire concernant le passé de l'enregistrement courant est contenue dans l'ensemble des règles actives.

Le second principe est réalisé au moyen de techniques d'implémentation efficaces qui supportent l'optimisation nécessaire. Une description détaillée de l'implémentation dépasse le cadre de ce mémoire (voir [3] pour une description complète). Nous en donnons succinctement ci-dessous les quelques idées principales :

- Les règles RUSSEL sont traduites dans un code interne optimisé qui peut être efficacement exécuté par une machine abtraite appropriée (par exemple, optimisation des évaluations de conditions *and/or*).
- Avant d'appliquer les règles actives à l'enregistrement courant, celui-ci est traité une première fois de façon à créer une table d'indirection optimisée contenant des pointeurs vers les différentes données de l'enregistrement. Cette technique assure un accès efficace à l'enregistrement courant qui, rappelons-le, a un format libre et une taille variable.
- Le système préserve à tout moment trois ensembles de règles : celles actives pour l'enregistrement courant, celles à activer pour l'enregistrement suivant et celles à activer à la fin de l'analyse. Chaque ensemble est représenté par une liste chaînée. Le passage d'un enregistrement au suivant est ainsi réalisé de façon directe en faisant glisser l'ensemble des règles actives pour l'enregistrement suivant vers l'ensemble des règles actives pour l'enregistrement courant, et en réinitialisant l'ensemble des règles actives pour l'enregistrement suivant à l'ensemble vide.
- Quand le traitement d'une règle est commencé, les paramètres effectifs sont copiés dans un emplacement qui est identique pour toutes les règles. Ceci permet un accès direct aux valeurs effectives des paramètres.

1.2 Langage RUSSEL

Une *déclaration* de règle en RUSSEL est composée d'un nom, d'un ensemble de paramètres formels, d'un ensemble de variables locales et d'une partie action. Une règle *active* est une *instanciation* d'une déclaration de règle avec des paramètres effectifs. Les règles actives sont exécutées l'une après l'autre : à tout moment il y a une seule règle en cours d'exécution. Des structures de contrôle classiques peuvent être utilisées pour écrire des actions conditionnelles, répétitives, etc. Les actions élémentaires incluent l'affectation, le déclenchement de règles et l'appel de routines externes. Cette dernière action permet aux programmeurs d'utiliser des procédures écrites dans un autre langage, comme le langage C, pour lire des input, écrire des output, accéder à des données complexes, etc. Les routines externes peuvent être appelées pour achever n'importe quelle caractéristique qui n'a rien à voir avec le traitement séquentiel du fichier d'audit.

Notons qu'une règle doit explicitement être redéclenchée si elle doit survivre pour l'analyse de l'enregistrement suivant car par défaut, une règle meurt après son déclenchement. L'idée est d'avoir un contrôle simple et direct sur la vie des règles et d'éviter un mécanisme complexe pour tuer des règles obsolètes.

1.2.1 Fonction *is_active*

Il peut se faire, dans certain cas, qu'une même règle soit déclenchée plusieurs fois avec les mêmes paramètres effectifs. C'est par exemple le cas si on cherche à détecter une séquence de plusieurs login. Chacun des login déclenchera une règle. Dès lors, le nombre de règles déclenchées peut croître très vite et finir par saturer le système.

C'est pourquoi la fonction booléenne *is_active* a été implémentée. Cette fonction permet de tester, avant de déclencher une règle, si celle-ci a déjà été préalablement déclenchée avec les mêmes paramètres effectifs par un autre enregistrement. On peut également tester si une règle a déjà été déclenchée avec une valeur quelconque des paramètres effectifs, ou même ne s'intéresser qu'à certaines des valeurs prises par les paramètres effectifs. Cette fonction renvoie un booléen valant *true* si la règle a déjà été activée; elle renvoie $false$ sinon.

L'utilisation de la fonction *is_active* permet de réduire le nombre de règles déclenchées et ainsi d'alléger l'exécution de l'analyse de l'audit trail.

Remarquons que l'implémentation de cette fonction a constitué "l'entrée en matière" de ce travail par une familiarisation avec ASAX et l'implémentation de son noyau.

1.2.2 Quelques exemples de programmes RUSSEL

Le but ici n'est pas de développer des patterns sophistiqués mais simplement de montrer comment RUSSEL traite des situations typiques dans l'analyse de fichiers d'audit.

Dans les exemples qui suivent, les appels de procédures externes sont notés en italique, les mots clés du langage RUSSEL sont notés en gras, "Evt", "Res", "Terminal", "Userid" et "Timestp" identifient des champs d'enregistrement de fichiers d'audit. Notons que la présence des champs est toujours supposée optionnelle: un champ non présent dans l'enregistrement a une valeur spéciale conventionnelle.

Détection d'une pénétration externe avec un même login

Le scénario qui doit être détecté est l'occurrence de "maxtime" échec(s) de login pendant une période de "duration" secondes par un même utilisateur. La première règle détecte un premier échec de login et déclenche une règle de décompte avec un temps d'expiration et l'identification de l'utilisateur. La seconde règle compte l'échec de login suivant; elle reste active jusqu'à ce que le temps fixé ait expiré ou jusqu'à ce que le compteur devienne 1.

```
rule Failed_login(maxtime,duration: integer);
begin
```

```
if Evt='login' and Res='failure' and is_unsecure(Terminal)
   -- > trigger off for_next
        Count_rule1(maxtime-1,Timestp+duration,Userid)
fi;
trigger off for_next Failed_login(maxtime,duration)
end

rule Count_rule1(countdown,expiration,suspectid: integer);
if Evt='login' and Res='failure'
   and is_unsecure(Terminal) and Timestp < expiration
   and Userid=suspectid
   -- > if countdown > 1
          -- > trigger off for_next
               Count_rule1(countdown-1,expiration,suspectid);
          countdown=1
          -- > SendMessage(''failed login's for'',suspectid)
        fi;
   Timestp >= expiration
   -- > skip;
   true
   -- > trigger off for_next
        Count_rule1(countdown,expiration,suspectid)
fi
```

Notons que la règle ne fait rien si le *time stamp* `Timestp` est plus grand que le temps d'expiration `expiration`. Ceci a pour effet de tuer la règle puisqu'un redéclenchement explicite est requis pour perpétuer la règle.

Détection d'une pénétration interne par un *masquerader*

Le scénario à détecter est une session d'utilisation suspecte, à savoir un login d'un utilisateur provenant d'un terminal, à un moment inhabituel pour cet utilisateur suivi d'une utilisation en dehors des habitudes de cet utilisateur. Un comportement habituel d'utilisateur est décrit par un profil préétabli. Un profil (très simplifié) est une table dans laquelle chaque entrée représente une commande et un nombre maximum d'occurrences admissible pour cette commande pendant un délai d'une heure (par exemple).

La première règle détecte un login depuis un terminal qui n'est pas habituel, à une heure qui ne l'est pas non plus. Elle déclenche ensuite une règle qui compare le profil de l'utilisateur suspect avec son profil habituel.

```
rule Masquerader()
begin
if Evt='login' and   not habitual_term(Userid,Terminal)
               and   not habitual_time(Userid,Timestp)
   -- > trigger off for_next watch(Userid,profileof(Userid));
fi;
trigger off for_next Masquerader()
```

```
end
```

La deuxième règle observe un utilisateur. Quand elle détecte la première occurrence d'une commande appartenant au profil de l'utilisateur, elle déclenche une règle de comptage pour cet utilisateur et cette commande et fournit à cette règle un temps d'expiration. Cette règle demeure active jusqu'au prochain login de l'utilisateur avec un terminal et un moment habituels.

```
rule watch( suspectid:integer;
            profile:table[cmnd:byte_string,maxtime:int])
if Userid=suspectid and Evt='login'
   and habitual_time(Userid,Timestp)
   and habitual_term(Userid,Terminal
   -- > skip
   Userid=suspectid and is_an_entry_in(Evt,profile)
   -- > begin
          maxtimes := select(Evt,profile) ;
          trigger off for_next
          count_rule3(suspectid,maxtimes,Timestp+3600,Evt);
          trigger off for_next watch(suspectid,profile)
       end;
   true
   -- > trigger off for_next watch(suspectid,profile)
fi
```

La troisième règle compte les occurrences de la commande "cmnd" effectuée par un utilisateur suspect et demeure active jusqu'à son temps d'expiration ou jusqu'à ce que le compteur devienne 1.

```
rule count_rule3( suspectid, countdown, expiration:integer;
                  cmnd:byte_string)
if Userid=suspectid and Evt=cmnd and Timestp<expiration
   -- > if countdown > 1
          -- > trigger off for_next
               count_rule3(suspectid,countdown-1,expiration,cmnd);
          countdown = 1
          -- > SendMessage(''unusual use for:'', suspectid)
       fi;
   Timestp >= expiration
   -- > skip;
   true
   -- > trigger off for_next
       count_rule3(suspectid,countdown,expiration,cmnd);
fi
```

Remarquons que d'autres exemples sont disponibles dans [1].

La présentation de RUSSEL faite plus haut et les quelques exemples ci-dessus nous montrent que RUSSEL est assez puissant pour supporter toute

requête dont on pourrait avoir besoin pour analyser un fichier d'audit. Toutefois, le langage peut apparaître un peu difficile à utiliser car il inclut un style de programmation quelque peu différent de celui auquel un programmeur non averti est habitué.

1.3 Raison d'être du langage LaDAA

Comme il est souligné dans [2] et comme nous venons de le voir, l'utilisation effective d'ASAX nécessite l'apprentissage du langage RUSSEL, langage dont le style de programmation peut être quelque peu difficile à maîtriser par un utilisateur non expert. RUSSEL utilise en effet un mélange de paradigmes impératif et à base de règles. C'est pourquoi il semblait intéressant de concevoir un langage de plus haut niveau permettant à l'utilisateur d'exprimer des requêtes plus déclaratives sur les fichiers d'audit. Le nouveau langage devrait donc permettre à l'utilisateur de spécifier de manière déclarative le scénario particulier qu'il souhaite détecter dans un fichier d'audit.

Dans cette optique, le fichier d'audit est considéré comme une séquence finie d'événements, un événement pouvant être lui-même défini comme une fonction qui, à chaque nom de champ d'enregistrement associe la valeur prise par ce champ dans cet enregistrement. Ainsi, la détection d'un scénario est vue comme la recherche d'une sous-séquence particulière dans une séquence d'événements donnée (que nous appellerons *séquence fondamentale*). Pour correspondre au scénario voulu, cette sous-séquence doit vérifier un ensemble de *conditions*. Plus précisément, chaque élément de la sous-séquence doit correspondre à un enregistrement dont les champs vérifient une certaine condition (à savoir prennent telles valeurs particulières).

Par exemple, on peut vouloir détecter une sous-séquence de trois événements remplissant une condition d'erreur de login. Une telle sous-séquence sera détectée si on trouve, dans le fichier d'audit, trois enregistrements correspondant à un login et dont le champ de *return-error* indique que la commande ne s'est pas passée correctement.

Remarquons que les événements de la sous-séquence recherchée peuvent apparaître de façon non consécutive dans la séquence de départ. En effet, d'autres événements que ceux recherchés peuvent se produire et venir s'intercaler entre ceux-ci.

1.4 Expressions Régulières

1.4.1 Définition

Notons Σ un ensemble de symboles, appelés *lettres*, et notons Σ^* l'ensemble de toutes les séquences finies de lettres issues de Σ, appelées *mots*. On appelle Σ un alphabet. La syntaxe abstraite d'une expression régulière est définie comme suit :

$$r ::= a \mid r.r \mid r + r \mid r^*$$

où a désigne une lettre quelconque dans Σ.

La sémantique d'une expression régulière sur l'alphabet Σ est donnée par une fonction f qui associe une expression régulière à un ensemble de mots de Σ^* :

$$f \ : \ RE \rightarrow \wp(\Sigma^*)$$

où RE désigne l'ensemble des expressions régulières et $\wp(\Sigma^*)$ désigne l'ensemble des parties de Σ^*, à savoir l'ensemble de tous les sous-ensembles de Σ^*.

On définit f récursivement sur la structure syntaxique d'une expression régulière de la façon suivante :

- $f(a) = \{a\}$
- $f(r_1.r_2) = \{w_1.w_2 \dagger w_1 \in f(r_1) \ \wedge \ w_2 \in f(r_2)\}$
- $f(r_1 + r_2) = f(r_1) \cup f(r_2)$
- $f(r^*) = \{w \dagger \exists n \in \mathbb{N} \ \exists w_1, \ldots, w_n \in f(r) \ : \ w = w_1. \ \ldots \ .w_n\}$

où a désigne une lettre quelconque de Σ et r, r_1, r_2 désignent des expressions régulières. Etant donné une expression régulière r, l'ensemble $f(r)$ est appelé le langage de r.

Par exemple, si $\Sigma = \{a, b\}$, nous avons que $f(a + b) = \{a, b\}$ et $f(ab^*) = \{a, ab, abb, abbb, \ldots\}$.

1.4.2 Langage LaDAA et expressions régulières

Le langage LaDAA se base en partie sur le concept des expressions régulières. L'alphabet Σ est, dans ce cas, constitué de l'ensemble de toutes les valeurs possibles élémentaires d'enregistrements d'un fichier d'audit. Ainsi, la recherche d'une sous-séquence correspond à spécifier une expression régulière à partir de cet alphabet.

Par rapport à la syntaxe des expressions régulières, nous appelons *condition simple* la construction élémentaire a, et *conditions élaborées* les autres constructions. En outre, nous appelons *séquence* la construction $r.r$ et *union* la construction $r + r$. Notons cependant que l'étoile de Kleene (r^*) n'est pas utilisée par le langage LaDAA[2]. En outre, le nouveau langage implémente une construction un peu particulière appelée *ensemble* que nous notons $\{r, r\}$. On pourrait définir la fonction f pour cette construction de la manière suivante :

$$f(\{r_1, r_2\}) = f \ \dagger \ interlaced(f, f(r_1), f(r_2)) \ = \ true$$

et la fonction *interlaced* se définit comme suit :

Soient A, B et C trois ensembles non vides.

Supposons que :

- $A = \{e_1, e_2, \ldots, e_n\}$,
- $B = \{e_{i_1}, e_{i_2}, \ldots, e_{i_m}\}$,
- $C = \{e_{j_1}, e_{j_2}, \ldots, e_{j_p}\}$.

[2]Une extension consisterait en l'introduction de l'étoile de Kleene le langage LaDAA.

Alors, $interlaced(A, B, C) = true$ *si*

- $1 \leq i_1 < i_2 < \ldots < i_m \leq n$
- $1 \leq j_1 < j_2 < \ldots < j_p \leq n$
- $\{i_1, i_2, \ldots, i_m\} \cap \{j_1, j_2, \ldots, j_p\} = \emptyset$

L'idée intuitive derrière un ensemble est la même que pour une séquence où l'ordre des éléments n'a pas d'importance.

Finalement, la syntaxe abstraite d'une expression régulière est modifiée dans le cadre de notre langage comme suit :

$$r ::= a \mid r.r \mid r + r \mid \{r, r\}$$

où a désigne une lettre quelconque dans Σ.

Chapitre 2

Description informelle de LaDAA

L'idée principale du langage LaDAA est de détecter une sous-séquence particulière dans une séquence[1] que nous appelons *séquence fondamentale.* En outre, la caractéristique principale du langage est d'être déclaratif. Ainsi, les propriétés particulières de la sous-séquence recherchée sont exprimées sous forme d'une requête déclarative que nous appelons *condition.*

Conditions simples

Le cas de base consiste en ce que nous appelons une *condition simple.* Une condition simple porte sur un ou plusieurs champs de l'enregistrement courant. Voici quelques exemples de conditions simples:

- `event='login'`
- `event='login' or event='chmod'`
- `(event='login' or event='chmod') and result='failure'`

Notons qu'une sous-séquence vérifiant une condition simple est composée d'un seul élément.

En outre, des variables peuvent apparaître dans les conditions simples. Remarquons que les variables du langage LaDAA sont toutes de type *string* et qu'elles sont toutes non définies au début de l'exécution. Pour des raisons de facilité lors de l'implémentation, elles doivent toutes être déclarées au début d'un programme LaDAA avant même la spécification de la condition.

Lorsqu'une variable apparaît dans une condition simple du genre `x=field`, deux cas sont possibles: soit la variable `x` est non définie et la condition simple est en fait une affectation du champ à la variable, soit la variable `x` possède une valeur et la condition simple est évaluée à *true* si cette valeur est égale à la valeur prise dans l'enregistrement courant par le champ intervenant dans la condition (elle est évaluée à $false$ sinon).

[1] Cette séquence correspond dans la pratique à la suite chronologique des enregistrements d'un fichier d'audit.

Si une variable apparaît dans une condition simple d'un autre genre (par exemple `x<field`), alors la variable `x` doit avoir reçu une valeur avant l'apparition de cette condition, faute de quoi une erreur surviendrait. Nous supposons dès lors que toute variable apparaissant dans ce genre de condition simple n'est pas non définie.

Conditions élaborées

En plus des conditions simples, le langage LaDAA permet de spécifier des *conditions élaborées*. Il existe trois types de conditions élaborées : les séquences, les unions et les ensembles.

Une *séquence* est composée de plusieurs conditions (simples ou élaborées) que l'on souhaite voir satisfaites dans un ordre précis par les éléments de la sous-séquence recherchée. Si nous notons les conditions simples par des lettres capitales, un exemple de séquence est : `sequ_of(A followed_by B followed_by B followed_by C)`. Cette condition sera satisfaite si on trouve une sous-séquence de quatre éléments dont le premier vérifie la condition `A`, le deuxième et le troisième la condition `B` et le quatrième la condition `C`. Remarquons bien que les différents éléments de la sous-séquence ne doivent pas nécessairement apparaître de manière consécutive dans la séquence fondamentale : d'autres éléments peuvent venir s'intercaler entre eux.

Une *union* est composée de plusieurs conditions (simples ou élaborées) dont on souhaite qu'au moins une soit satisfaite par le(s) élément(s) de la sous-séquence recherchée. En notant les conditions simples par des lettres capitales, un exemple d'union est : `union_of(A orelse B orelse sequ_of(C followed_by D))`. Cette condition sera satisfaite si on trouve dans la séquence fondamentale : soit une sous-séquence composée d'un élément vérifiant la condition `A` ou la condition `B`, soit une sous-séquence composée de deux éléments dont le premier vérifie la condition `C` et le second la condition `D`.

Un *ensemble* est composé de plusieurs conditions (simples ou élaborées) que l'on souhaite voir satisfaites dans un ordre quelconque par les éléments de la sous-séquence recherchée. Ce type de condition est le plus complexe des types de conditions élaborées. Un exemple d'ensemble où les conditions simples sont notées par des lettres capitales est : `set_of(A coupled_with B coupled_with C)`. Cette condition sera satisfaite si on trouve dans la séquence fondamentale une sous-séquence de trois éléments dont chacun vérifie une des trois conditions simples `A`, `B` ou `C`, et vérifiant chacun une condition simple distincte.

Remarquons que nous pourrions par exemple avoir la condition élaborée `set_of(C coupled_with C coupled_with C)`. Nous verrons, dans les exemples donnés plus loin, que cette formulation fournira les mêmes résultats que la formulation "séquentielle" équivalente (`sequ_of(C followed_by C followed_by C)`) mais qu'elle sera moins efficace à l'exécution.

Conditions prédéfinies

Pour des raisons de facilité pour l'utilisateur, le langage LaDAA permet de définir des conditions typiques prédéfinies au début de toute requête. On peut

ainsi "baptiser" une condition et utiliser ce nom dans la requête en lieu et place de la condition. Ceci permet d'alléger considérablement l'écriture de certaines requêtes. Un exemple de définition de condition prédéfinie est : `sequ toto is event='login' and result='error' and x=userid`. Cette condition spécifie une recherche d'erreur de login où l'on veut retenir le nom de l'utilisateur. Un exemple complet de requête est présenté ci-dessous :

```
with_declarations
   var   string x
   sequ  toto is event='login' and result='error' and x=userid
find first sequ_of(toto followed_by toto followed_by toto)
```

Cet exemple est une traduction en langage LaDAA de l'exemple de programme RUSSEL à propos de détection d'une pénétration externe avec un même login donné à la page 13. Notons toutefois que la notion d'expiration des règles n'est pas traduite dans le langage LaDAA. Plutôt que de tuer les règles après un certain laps de temps, nous verrons lors de son implémentation (par traduction en langage RUSSEL) que nous avons choisi de ne déclencher, grâce à l'utilisation de la fonction *is_active*, que le minimum de règles nécessaires à la recherche de la sous-séquence voulue.

La comparaison des deux exemples de programmes en LaDAA et en RUSSEL montre clairement la différence de style de programmation : déclaratif pour le langage LaDAA, impératif et à base de règles pour le langage RUSSEL.

Avertissement

Le terme *séquence* est utilisé dans ce travail avec plusieurs significations qu'il est important de bien distinguer :

- le terme *séquence fondamentale* désigne la séquence chronologique des enregistrements contenus dans un fichier d'audit et dans laquelle on souhaite extraire une sous-séquence particulière,
- le terme *séquence prédéfinie* désigne une condition prédéfinie déclarée au début d'un programme LaDAA et qui peut ensuite être utilisée dans la requête en lieu et place de la dite condition,
- le terme *condition élaborée de type séquence* (ou plus simplement *séquence*) désigne la construction du langage LaDAA représentant une condition (élaborée de type séquence) que la sous-séquence recherchée doit vérifier.

Chapitre 3

Syntaxe du langage LaDAA

Nous donnons ci-dessous les syntaxes abstraite et concrète du langage LaDAA. Nous donnons tout d'abord nos domaines syntaxiques, puis définissons la syntaxe abstraite de notre langage. Nous expliquons ensuite chacune des constructions de la syntaxe abstraite et donnons la syntaxe concrète correspondante. Nous finissons par un récapitulatif de la syntaxe concrète.

3.1 Domaines syntaxiques et syntaxe abstraite

Nous utilisons les domaines syntaxiques suivants [1] :

c	$\in$	Constant	O	$\in$	Opredef
fd	$\in$	Field	I	$\in$	Identificateur
var	$\in$	Variable	E	$\in$	Expression
m	$\in$	Mode	C	$\in$	Condition
r	$\in$	Restriction	S	$\in$	Condition_elaboree
a	$\in$	Action	R	$\in$	Recherche
			DV	$\in$	Declaration_Variable
P	$\in$	Program	DS	$\in$	Declaration_Sequence

L'entièreté de la syntaxe abstraite du langage LaDAA est reprise ci-dessous.

P	=	$prog(\mathrm{DV}, \mathrm{DS}, \mathrm{R})$
DV	=	$\epsilon \mid DecVar(var) \mid DeclV(\mathrm{DV}_1, \mathrm{DV}_2)$
DS	=	$\epsilon \mid DecSeq(var, \mathrm{S}) \mid DeclS(\mathrm{DS}_1, \mathrm{DS}_2)$
R	=	$rech(S, m, r, a)$
S	=	$\mathrm{C} \mid var \mid follow(\mathrm{S}_1, \mathrm{S}_2) \mid union(\mathrm{S}_1, \mathrm{S}_2) \mid couple(\mathrm{S}_1, \mathrm{S}_2)$
C	=	$\mathrm{E} \mid and(\mathrm{C}_1, \mathrm{C}_2) \mid or(\mathrm{C}_1, \mathrm{C}_2)$
E	=	$c \mid I \mid op(\mathrm{E}_1, \mathrm{O}, \mathrm{E}_2)$
I	=	$fd \mid var$

En quelques mots, un programme est constitué de déclarations d'une part, et d'une recherche particulière d'autre part. Les déclarations sont divisées en déclarations de variables et déclarations de séquences prédéfinies typiques. Une

[1] Nous conseillons au lecteur non habitué à ce formalisme de se reporter au chapitre 13 de [4] pour une introduction et une familiarisation avec les notations utilisées.

recherche est définie par une sous-séquence qui doit remplir soit une condition simple, soit une condition élaborée. Nous reprenons à présent l'une après l'autre chacune des constructions syntaxiques.

3.2 Explication intuitive de chaque construction syntaxique et syntaxe concrète

3.2.1 Programme

La syntaxe abstraite d'un programme est la suivante:

$$\mathrm{P} = prog(\mathrm{DV}, \mathrm{DS}, \mathrm{S})$$

Un programme LaDAA est constitué de deux parties principales: une partie contenant les déclarations de variables et de séquences prédéfinies et une partie contenant la spécification de la condition devant être satisfaite par la sous-séquence recherchée. La syntaxe concrète d'un programme est la suivante:

$$\begin{array}{lll} < program > & ::= & \underline{with\_declarations} < declaration > \underline{find} < search > \\ < declaration > & ::= & < declaration\_variable > \hookleftarrow < definition\_sequence > \end{array}$$

3.2.2 Déclarations

Il existe deux types de déclarations dans le langage LaDAA: les déclarations de variables et les déclarations de séquences prédéfinies.

Déclarations de variables

La syntaxe abstraite des déclarations de variables est la suivante:

$$\mathrm{DV} = \epsilon \mid DecVar(var) \mid DeclV(\mathrm{DV}_1, \mathrm{DV}_2)$$

On peut déclarer, dans le langage LaDAA, un nombre fini quelconque de variables, ou bien ne pas déclarer de variables du tout. Remarquons bien que les variables ont ici un rôle particulier différent du rôle habituel d'une variable dans un langage impératif classique tel que le langage C ou le langage Pascal, à savoir qu'*une variable ne peut recevoir qu'une seule fois une valeur* au cours de l'exécution d'un programme, à savoir la première fois que cette variable apparaît dans une condition simple du genre `x=field`. En outre, les variables ne sont *jamais* affectées d'une valeur particulière mais *toujours* d'un nom de champ. C'est pourquoi on ne peut définir que des variables de type *string*, la déclaration de variables d'un autre type étant inutile dans le langage LaDAA puisque ces variables ne pourraient jamais être affectées d'une valeur.

Les variables ont été introduites dans le langage LaDAA afin de pouvoir "capturer" la valeur d'un champ dans le record courant et comparer par après cette valeur à la valeur prise par ce champ dans un autre record. Cette astuce permet, en quelque sorte, un passage de paramètres. La syntaxe concrète des

déclarations de variables est la suivante :

$$
\begin{array}{lcl}
< declaration\_variable > & ::= & \underline{novars} \mid \underline{var} \ < dec\_var > \\
< dec\_var > & ::= & \underline{string} \ < var\_group > \\
 & \mid & < dec\_var > \hookleftarrow \\
 & & \underline{string} \ < var\_group > \\
< var\_group > & ::= & < identifier > \\
 & \mid & < var\_group > \ \square \ < identifier >
\end{array}
$$

Déclarations de séquences prédéfinies

La syntaxe abstraite des déclarations de séquences prédéfinies[2] est la suivante :

$$\mathrm{DS} = \epsilon \mid DecSeq(var, \mathrm{S}) \mid DeclS(\mathrm{DS}_1, \mathrm{DS}_2)$$

Afin d'alléger le code, il est possible de définir un nombre fini quelconque de *séquences prédéfinies* et de leur donner à chacune un nom identifiant que l'on appellera dans la suite du programme en lieu et place de la condition en question. En fait, les séquences prédéfinies sont des *conditions* que la sous-séquence recherchée doit vérifier.
La syntaxe concrète des déclarations de séquences prédéfinies est la suivante :

$$
\begin{array}{lcl}
< definition\_sequence > & ::= & \underline{nosequ} \mid < def\_seq > \\
< def\_seq > & ::= & \underline{sequ} \ < identifier > \ \underline{is} < condition > \\
 & \mid & < def\_seq > \hookleftarrow \\
 & & \underline{sequ} \ < identifier > \ \underline{is} < condition >
\end{array}
$$

Nous verrons plus loin quelle est la syntaxe des conditions.

3.2.3 Recherche de sous-séquence

La syntaxe abstraite d'une recherche de sous-séquence est la suivante :

$$\mathrm{R} = rech(S, m, r, a)$$

Outre la spécification de la condition particulière que la sous-séquence recherchée doit vérifier, le langage LaDAA comprend un mode qui indique quelle(s) sous-séquence(s) l'utilisateur désire trouver : la première, la plus courte (en durée dans le temps), n'importe laquelle, ou toutes. En outre, une restriction sur la durée maximale entre les différents événements de la sous-séquence peut être imposée aux sous-séquences recherchées.

De plus, lorsqu'une exécution de programme trouve dans le fichier d'audit une sous-séquence recherchée par l'utilisateur, il convient d'en avertir celui-ci par l'impression d'un message (choisi par l'utilisateur). L'apparition de ce message indique à l'utilisateur que le fichier contient bien une sous-séquence qui l'intéresse. On pourrait imaginer d'autres actions que l'impression d'un message lors de la découverte d'une sous-séquence, mais nous avons retenu cette

[2] Cette construction syntaxique aurait tout aussi bien pu s'appeler *condition prédéfinie*.

solution dans le cadre de ce mémoire. La syntaxe concrète d'une recherche de sous-séquence est la suivante :

$$
\begin{array}{lll}
< search > & ::= & < mode > < find > \\
< mode > & ::= & \underline{first} \mid \underline{shortest} \mid \underline{any} \mid \underline{all} \\
< find > & ::= & < cond\_elab > \\
 & & \underline{with} < restriction > \\
 & & \underline{performing} < action > \\
< restriction > & ::= & \underline{norestriction} \\
 & \mid & \underline{duration} < expression > \\
< action > & ::= & \underline{print} < string >
\end{array}
$$

Conditions

La syntaxe abstraite des conditions spécifiées pour la recherche de sous-séquences est la suivante :

$$S = C \mid var \mid follow(S_1, S_2) \mid union(S_1, S_2) \mid couple(S_1, S_2)$$

La sous-séquence recherchée doit soit remplir une condition simple, soit être une séquence prédéfinie ou soit remplir une condition élaborée. Nous distinguons trois types de conditions élaborées : les séquences, les unions et les ensembles. Nous appelons *séquence* une suite de plusieurs conditions (simples ou élaborées) que l'on désire rencontrer dans un ordre précis dans le fichier d'audit. Nous appelons *union* un groupe de plusieurs conditions (simples ou élaborées) dont chacune suffit à ce que la condition élaborée de type union soit satisfaite. Enfin, nous appelons *ensemble* un ensemble (et plus précisément un multi-ensemble au sens mathématique) de conditions (simples ou élaborées); l'ordre n'a dès lors pas d'importance. La syntaxe concrète des conditions élaborées est la suivante :

$$
\begin{array}{lll}
< cond\_elab > & ::= & < condition > | < identifier > \\
 & \mid & \underline{sequ\_of}(< cond\_elab > [\underline{followed\_by} < cond\_elab >]^*) \\
 & \mid & \underline{set\_of}(< cond\_elab > [\underline{coupled\_with} < cond\_elab >]^*) \\
 & \mid & \underline{union\_of}(< cond\_elab > [\underline{orelse} < cond\_elab >]^*)
\end{array}
$$

Conditions simples

La syntaxe abstraite des conditions simples spécifiées pour la recherche de sous-séquences est la suivante :

$$C = E \mid and(C_1, C_2) \mid or(C_1, C_2)$$

Les conditions simples peuvent soit être une expression ou soit combiner plusieurs conditions simples par un "and" ou un "or". Généralement, cette expression sera une comparaison entre un nom de champ et une valeur particulière. Remarquons que notre syntaxe concrète fixe des priorités sur la façon de combiner les différentes conditions. La syntaxe concrète des conditions est la suivante :

$$
\begin{array}{lll}
< condition > & ::= & < condi > \\
 & | & < condition > \underline{and} < condition > \\
< condi > & ::= & < cond > \\
 & | & (< cond > [\ \underline{or}\ < cond >]^*) \\
< cond > & ::= & < expression > < op > < expression > \\
< op > & ::= & <|>|! =|=|<=|>=
\end{array}
$$

Expressions

La syntaxe abstraite des expressions est la suivante :

$$E = c \mid I \mid op(E_1, O, E_2)$$

Une expression, dans le langage LaDAA, peut être soit une constante, soit un nom de champ ou de variable, soit une opération entre deux expressions. La syntaxe concrète des expressions est la suivante :

$$
\begin{array}{lll}
< expression > & ::= & < constant > \\
 & | & < identifier > \\
 & | & (\ < expression >\ ) \\
 & | & < expression >< bin\_op >< expression > \\
 & | & < un\_op >< expression > \\
 & | & < call\_to\_predefined\_procedure > \\
< bin\_op > & ::= & + \mid - \mid * \mid div \mid mod \\
< un\_op > & ::= & -
\end{array}
$$

Notons que le langage LaDAA permet d'intégrer dans les expressions toutes les fonctions prédéfinies actuellement par ASAX, comme par exemple la fonction *match* qui est définie comme suit :

$match(RE, string)$ = 1 si le *string* correspond avec l'expression régulière RE
= 0 sinon

Par exemple :

- $match('Failed', 'Failed\ login') = 1$,
- $match('unknown', 'babbage : syslog : Connection\ unknown') = 1$
- $match('Secure', 'Security') = 0$.

3.3 Récupitulatif de la syntaxe concrète

$$
\begin{array}{lcl}
< program > & ::= & \underline{with\_declarations} < declaration > \underline{find} < search > \\
< declaration > & ::= & < declaration\_variable > \hookleftarrow \\
 & & < definition\_sequence > \\
< declaration\_variable > & ::= & \underline{novars} \mid \underline{var} \ < dec\_var > \\
< dec\_var > & ::= & \underline{string} \ < var\_group > \\
 & \mid & < dec\_var > \hookleftarrow \\
 & & \underline{string} \ < var\_group > \\
< var\_group > & ::= & < identifier > \\
 & \mid & < var\_group > \ \square \ < identifier > \\
< definition\_sequence > & ::= & \underline{nosequ} \mid < def\_seq > \\
< def\_seq > & ::= & \underline{sequ} \ < identifier > \ \underline{is} < condition > \\
 & \mid & < def\_seq > \hookleftarrow \\
 & & \underline{sequ} \ < identifier > \ \underline{is} < condition > \\
< search > & ::= & < mode > < find > \\
< mode > & ::= & \underline{first} \mid \underline{shortest} \mid \underline{any} \mid \underline{all} \\
< find > & ::= & < cond\_elab > \\
 & & \underline{with} < restriction > \\
 & & \underline{performing} < action > \\
< restriction > & ::= & \underline{norestriction} \\
 & \mid & \underline{duration} < expression > \\
< action > & ::= & \underline{print} < string > \\
< cond\_elab > & ::= & \underline{sequ\_of}(< cond\_elab > \ [\underline{followed\_by} \ < cond\_elab >]^*) \\
 & \mid & \underline{set\_of}(< cond\_elab > \ [\underline{coupled\_with} \ < cond\_elab >]^*) \\
 & \mid & \underline{union\_of}(< cond\_elab > \ [\underline{orelse} \ < cond\_elab >]^*) \\
 & \mid & < condition > \mid < identifier > \\
< condition > & ::= & < condi > \\
 & \mid & < condition > \underline{and} < condition > \\
< condi > & ::= & < cond > \\
 & \mid & (< cond > [\ \underline{or} \ < cond >]^*) \\
< cond > & ::= & < expression > < op > < expression > \\
< op > & ::= & < \mid > \mid ! = \mid = \mid < = \mid > = \\
< expression > & ::= & < constant > \\
 & \mid & < identifier > \\
 & \mid & (\ < expression > \ ) \\
 & \mid & < expression > < bin\_op > < expression > \\
 & \mid & < un\_op > < expression > \\
 & \mid & < call\_to\_predefined\_procedure > \\
< bin\_op > & ::= & + \mid - \mid * \mid div \mid mod \\
< un\_op > & ::= & -
\end{array}
$$

Chapitre 4

Sémantique du langage LaDAA

Nous donnons ici la sémantique de notre langage sous forme dénotationnelle[1]. Nous définissons tout d'abord les domaines sémantiques, puis donnons une fonction sémantique pour chacune des constructions de notre syntaxe abstraite. Nous verrons ici que le langage LaDAA est déclaratif en ce sens que la sémantique d'un programme est de spécifier sous quelles conditions on peut trouver une sous-séquence particulière dans la séquence fondamentale. Il n'est nullement expliqué comment l'exécution d'un programme procède pour trouver effectivement cette sous-séquence.

4.1 Domaines sémantiques

4.1.1 Définition

$$
\begin{array}{lcl}
\mathbb{F}\text{ield} & = & \mathbb{I}\text{dentifier} \\
\$\text{tring} & = & \mathbb{B}\text{yte}^* \\
\mathbb{E}\text{vent} & = & \mathbb{F}\text{ield} \rightarrow \$\text{tring} + \{not\_pertinent\} \\
\$\text{equence} & = & \mathbb{E}\text{vent}^* \\
\mathbb{E}\text{nv}\mathbb{V}\text{ar} & = & \mathbb{I}\text{dentifier} \rightarrow \$\text{tring} + \{undef\} + \{noval\} \\
\mathbb{E}\text{nv}\$\text{eq} & = & \mathbb{I}\text{dentifier} \rightarrow (\mathbb{E}\text{nv}\mathbb{V}\text{ar} \rightarrow \$\text{equence} \rightarrow \$\text{equence} \rightarrow \mathbb{B}\text{ool}) + \{not\_def\}
\end{array}
$$

4.1.2 Explication informelle des domaines sémantiques

Le domaine principal est celui des événements (IEvent) qui modélise les enregistrements d'un fichier d'audit. Nous avons dès lors choisi de les représenter par une fonction qui, à chaque champ de l'enregistrement, associe soit la valeur prise par ce champ dans l'enregistrement du fichier d'audit (de type string), soit la valeur *not_pertinent* si ce champ n'est pas défini dans l'enregistrement considéré.

[1]Nous conseillons au lecteur non habitué à ce formalisme de se reporter au chapitre 13 de [4] pour une introduction et une familiarisation avec les notations utilisées.

Le domaine séquence ($equence) modélise les séquences et sous-séquences d'événements présents dans un fichier d'audit. Elles sont dès lors modélisées par une suite de 𝔼vent.

Le domaine string ($tring) que nous considérons ici est défini, tout comme dans ASAX, comme une simple suite de bytes. Cela permet de représenter toutes les valeurs possibles aussi bien de chaînes de caractères que d'entiers.

Les domaines 𝔼nv𝕍ar et 𝔼n$eq modélisent respectivement les environnements des variables et des sous-séquences prédéfinies, et sont définis comme des fonctions qui, à chaque identifiant (représentant une variable ou une séquence prédéfinie), associent la valeur pertinente qui lui correspond, à savoir un string dans le cas de variables, et une fonction retournant un booléen dans le cas des séquences prédéfinies (cette fonction prendra tout son sens lorsque nous définirons la sémantique des conditions élaborées).

4.2 Fonctions sémantiques

Pour chacune des constructions de la syntaxe abstraite, nous rappelons sa syntaxe puis donnons la signature et la définition de la fonction sémantique qui lui donne son sens.

4.2.1 Identificateur

Les noms de champ d'enregistrement et de variables constituent les identificateurs. C'est la fonction $\mathcal{I}$ qui donne la sémantique des identificateurs. Dans le cas d'un nom de champ, l'événement correspondant au record courant nous donne sa valeur. Dans le cas d'un nom de variable, c'est l'environnement des variables qui nous renvoie sa valeur.

Rappel de la syntaxe abstraite

I = *fd* | *var*

Signature de la fonction

$\mathcal{I}$: Identificateur → 𝔼vent → 𝔼nv𝕍ar → $tring

Définition de la fonction

$$\begin{aligned} \mathcal{I}[\![fd]\!]\, e\, v &= e[\![fd]\!] \\ \mathcal{I}[\![var]\!]\, e\, v &= v[\![var]\!] \end{aligned}$$

4.2.2 Expression

Les expressions sont soit des constantes, soit des identificateurs, soit des expressions composées de plusieurs expressions mises en relation par une opération prédéfinie.

Les opérations prédéfinies du langage LaDAA sont les mêmes que celles du langage RUSSEL, à savoir : $+, -, *, div, mod, <, >, <=, >=, ! =, =$. Nous supposons qu'il existe une fonction $\mathcal{O}$ qui donne le sens des opérations prédéfinies.

C'est la fonction $\mathcal{E}$ qui donne la sémantique des expressions. Une constante est évaluée par appel à la fonction $\mathcal{V}$ que nous supposons être la fonction qui donne la sémantique des constantes. Un identificateur est évalué par appel à la fonction $\mathcal{I}$ et une expression composée est évaluée par le résultat de l'application de la fonction $\mathcal{O}$ aux évaluations respectives de chaque composante de l'expression par la fonction $\mathcal{E}$.

Rappel de la syntaxe abstraite

$\mathrm{E} = c \mid \mathrm{I} \mid op(\mathrm{E}_1, \mathrm{O}, \mathrm{E}_2)$

Signature de la fonction

$\mathcal{E}$: Expression $\rightarrow$ $\mathbb{E}$vent $\rightarrow$ $\mathbb{E}$nv$\mathbb{V}$ar $\rightarrow$ $\mathbb{S}$tring + $\mathbb{B}$ool

Définition de la fonction

$$\begin{array}{lcl} \mathcal{E}[\![c]\!]\ e\ v & = & \mathcal{V}[\![c]\!] \\ \mathcal{E}[\![I]\!]\ e\ v & = & \mathcal{I}[\![I]\!]\ e\ v \\ \mathcal{E}[\![op(\mathrm{E}_1, \mathrm{O}, \mathrm{E}_2)]\!]\ e\ v & = & \mathcal{O}[\![\mathrm{O}]\!]\ (\mathcal{E}[\![\mathrm{E}_1]\!]\ e\ v)\ (\mathcal{E}[\![\mathrm{E}_2]\!]\ e\ v) \end{array}$$

4.2.3 Condition simple

Les conditions simples sont soit une expression, soit la combinaison de plusieurs expressions par un *and* ou un *or*. C'est la fonction $\mathcal{C}$ qui donne la sémantique des conditions. L'évaluation d'une expression est faite par appel à la fonction $\mathcal{E}$, l'évaluation des autres expressions combine (par un *et logique* ou un *ou logique*) l'évaluation de chacune de leurs composantes par la fonction $\mathcal{C}$.

Rappel de la syntaxe abstraite

$\mathrm{C} = \mathrm{E} \mid and(\mathrm{C}_1, \mathrm{C}_2) \mid or(\mathrm{C}_1, \mathrm{C}_2)$

Signature de la fonction

$\mathcal{C}$: Condition $\rightarrow$ $\mathbb{E}$vent $\rightarrow$ $\mathbb{E}$nv$\mathbb{V}$ar $\rightarrow$ $\mathbb{B}$ool

Définition de la fonction

$$\begin{array}{lcl} \mathcal{C}[\![\mathrm{E}]\!]\ e\ v & = & \mathcal{E}[\![\mathrm{E}]\!]\ e\ v \\ \mathcal{C}[\![and(\mathrm{C}_1, \mathrm{C}_2)]\!]\ e\ v & = & (\mathcal{C}[\![\mathrm{C}_1]\!]\ e\ v)\ \underline{et}\ (\mathcal{C}[\![\mathrm{C}_2]\!]\ e\ v) \\ \mathcal{C}[\![or(\mathrm{C}_1, \mathrm{C}_2)]\!]\ e\ v & = & (\mathcal{C}[\![\mathrm{C}_1]\!]\ e\ v)\ \underline{ou}\ (\mathcal{C}[\![\mathrm{C}_2]\!]\ e\ v) \end{array}$$

4.2.4 Condition élaborée

Une construction syntaxique de type condition élaborée correspond à la condition qu'une sous-séquence recherchée doit vérifier. Rappelons que cette condition peut soit être simple, soit être prédéfinie ou soit être une condition élaborée du type séquence, union ou ensemble. C'est la fonctions $\mathcal{S}$ qui leur donne leur sémantique. Elle prend pour arguments, en plus de la construction syntaxique, l'environnement des variables, la séquence fondamentale dans laquelle on cherche une sous-séquence, la sous-séquence recherchée ainsi que l'environnement des séquences prédéfinies. Elle renvoie un booléen valant *true* si, les environnements de variables et de séquences prédéfinies étant donnés, on peut trouver, dans la séquence fondamentale, une sous-séquence vérifiant la condition spécifiée. Elle renvoie $false$ sinon.

Une condition simple est évaluée à *true* si la sous-séquence est constituée d'un élément de la séquence fondamentale qui vérifie cette condition simple (par rapport à l'évaluation de la condition par la fonction $\mathcal{C}$).

Une séquence prédéfinie est évaluée par appel à la fonction qui lui est associée dans l'environnement des séquences prédéfinies (on comprend dès lors le choix de la définition du domaine $\mathbb{E}$nv$\$$eq).

Une condition élaborée de type séquence est évaluée à *true* si on peut décomposer la sous-séquence recherchée en deux *sous-sous-séquences* non vides qui soient telles que l'on puisse également trouver dans la séquence fondamentale deux sous-séquences de sorte que :

- la première sous-sous-séquence est contenue dans la première sous-séquence de la séquence fondamentale et vérifie la première composante de la condition élaborée,
- la seconde sous-sous-séquence est contenue dans la seconde sous-séquence de la séquence fondamentale et vérifie la seconde composante de la condition élaborée.

Une condition élaborée de type union est évaluée à *true* si au moins une de ses composantes est évaluée à *true* par la fonction $\mathcal{S}$ (avec les mêmes paramètres).

Finalement, une condition élaborée de type ensemble est évaluée à *true* si on peut trouver deux *sous-sous-séquences* non vides de la sous-séquence recherchée qui soient *entrelacées* (voir le paragraphe suivant pour une définition précise du concept d'entrelacement) dans la séquence fondamentale, et qui vérifient chacune une composante différente de la condition élaborée.

Rappel de la syntaxe abstraite

$S = C \mid var \mid follow(S_1, S_2) \mid union(S_1, S_2) \mid couple(S_1, S_2)$

Signature de la fonction

$\mathcal{S}$: Sequence $\rightarrow$ $\mathbb{E}$nv$\mathbb{V}$ar $\rightarrow$ $\$$equence $\rightarrow$ $\$$equence $\rightarrow$ $\mathbb{E}$nv$\$$eq $\rightarrow$ $\mathbb{B}$ool

Définition de la fonction

$$
\begin{array}{lcll}
\mathcal{S}[\![\mathrm{C}]\!]\ v\ s\ <e>\ q & = & true & \underline{si}\ \ s = e.s' \\
 & & & \underline{et}\ \ C[\![\mathrm{C}]\!]\ e\ v = true \\
 & = & true & \underline{si}\ \ s = e'.s' \\
 & & & \underline{et}\ \ \mathcal{S}[\![\mathrm{C}]\!]\ v\ s'\ <e>\ q = true \\
 & = & false & (sinon) \\
 & & & \\
\mathcal{S}[\![var]\!]\ v\ s\ ss\ q & = & & q[\![var]\!]\ v\ s\ ss \\
 & & & \\
\mathcal{S}[\![follow(\mathrm{S}_1,\mathrm{S}_2)]\!]\ v\ s\ ss\ q & = & true & \underline{si}\ \ \exists s_1, s_2, ss_1, ss_2 \in \mathbb{E}\mathrm{vent}^+ \\
 & & & tq\ \ s = s_1.s_2\ \underline{et}\ ss = ss_1.ss_2 \\
 & & & \underline{et}\ \ \mathcal{S}[\![\mathrm{S}_1]\!]\ v\ s_1\ ss_1\ q = true \\
 & & & \underline{et}\ \ \mathcal{S}[\![\mathrm{S}_2]\!]\ v\ s_2\ ss_2\ q = true \\
 & = & false & (sinon) \\
 & & & \\
\mathcal{S}[\![union(\mathrm{S}_1,\mathrm{S}_2)]\!]\ v\ s\ ss\ q & = & true & \underline{si}\ \ \mathcal{S}[\![\mathrm{S}_1]\!]\ v\ s\ ss\ q = true \\
 & & & \underline{ou}\ \ \mathcal{S}[\![\mathrm{S}_2]\!]\ v\ s\ ss\ q = true \\
 & = & false & (sinon) \\
 & & & \\
\mathcal{S}[\![couple(\mathrm{S}_1,\mathrm{S}_2)]\!]\ v\ s\ ss\ q & = & true & \underline{si}\ \ \exists ss_1, ss_2 \in \mathbb{E}\mathrm{vent}^+ \\
 & & & \underline{et}\ \ interlace(ss, ss_1, ss_2) \\
 & & & \underline{et}\ \ \mathcal{S}[\![\mathrm{S}_1]\!]\ v\ s\ ss_1\ q = true \\
 & & & \underline{et}\ \ \mathcal{S}[\![\mathrm{S}_2]\!]\ v\ s\ ss_2\ q = true \\
 & = & false & (sinon)
\end{array}
$$

Définition de la fonction $interlace(ss, ss_1, ss_2)$

Cette fonction indique que les sous-séquences ss_1 et ss_2 sont *entrelacées* dans la séquence ss. Plus précisément, ceci signifie que :

- $s = (e_1, \ldots, e_n)$
- $ss_1 = (e_{i_1}, \ldots, e_{i_m})$
- $ss_2 = (e_{j_1}, \ldots, e_{j_p})$

avec :

- $1 \leq i_1 < \ldots < i_m \leq n$
- $1 \leq j_1 < \ldots < j_p \leq n$
- $\{i_1, \ldots, i_m\} \bigcap \{j_1, \ldots, j_p\} = \emptyset$

Par exemple, si la séquence ss est "*accababcaca*", alors on peut dire que les sous-séquences "*aaac*" et "*cbb*" sont entrelacées dans ss.

4.2.5 Déclaration de variables

Les variables sont déclarées globalement au début de tout programme. Leur déclaration modifie l'environnement des variables en fixant celles qui sont utilisées par le programme et celles qui ne le sont pas. C'est la fonction $\mathcal{D}v$ qui donne la sémantique des déclarations de variables. Cette fonction prend pour argument

un environnement de variables et retourne un environnement de variables mis-à-jour. S'il n'y a aucune déclaration de variables, l'environnement des variables retourné est celui donné en entrée. Sinon, l'environnement des variables est modifié de la façon suivante : chaque identificateur présent dans la déclaration est associé dans le nouvel environnement à la valeur *noval*.

Rappel de la syntaxe abstraite

$\mathrm{DV} = \epsilon \mid DecVar(var) \mid DeclV(\mathrm{DV}_1, \mathrm{DV}_2)$

Signature de la fonction

$\mathcal{D}v$: Declaration_variable $\rightarrow$ $\mathbb{E}$nv$\mathbb{V}$ar $\rightarrow$ $\mathbb{E}$nv$\mathbb{V}$ar

Définition de la fonction

$$\begin{array}{lcl} \mathcal{D}v[\![\epsilon]\!]\ v & = & v \\ \mathcal{D}v[\![DecVar(var)]\!]\ v & = & v[var/noval] \\ \mathcal{D}v[\![DeclV(\mathrm{DV}_1, \mathrm{DV}_2)]\!]\ v & = & \mathcal{D}v[\![\mathrm{DV}_2]\!]\ (\mathcal{D}v[\![\mathrm{DV}_1]\!]v) \end{array}$$

4.2.6 Déclaration de séquences

La possibilité est offerte à l'utilisateur de déclarer des conditions de sous-séquences prédéfinies au début de tout programme. Ceci permet une manipulation plus souple du langage en évitant de réécrire à chaque fois, dans une recherche complexe, une condition typique recherchée par l'utilisateur.

Remarquons que l'environnement des séquences (il serait même plus juste de parler d'environnement des conditions de sous-séquences) ainsi défini associe, à chaque nom de séquence prédéfinie, une fonction qui prend en argument une condition, une séquence fondamentale, une sous-séquence, ainsi qu'un environnement de variables, et renvoie un booléen indiquant si, l'environnement des variables étant donné, la sous-séquence vérifie, au sein de la séquence fondamentale, la dite condition.

C'est la fonction $\mathcal{D}s$ qui donne la sémantique des déclarations de séquences prédéfinies. Cette fonction prend pour argument un environnement de séquences prédéfinies et retourne un environnement de séquences prédéfinies mis-à-jour. S'il n'y a aucune déclaration de séquences prédéfinies, l'environnement des séquences prédéfinies retourné est celui donné en entrée. Sinon, l'environnement des séquences prédéfinies est modifié de la façon suivante : chaque identificateur présent dans la déclaration est associé dans le nouvel environnement à la fonction définissant la séquence prédéfinie en question.

Rappel de la syntaxe abstraite

$\mathrm{DS} = \epsilon \mid DecSeq(var) \mid DeclS(\mathrm{DS}_1, \mathrm{DS}_2)$

Signature de la fonction

$\mathcal{D}s$: Declaration_sequence $\rightarrow$ $\mathbb{E}$nv$\$$eq $\rightarrow$ $\mathbb{E}$nv$\$$eq

Définition de la fonction

$$\begin{array}{lcl} \mathcal{D}s[\![\epsilon]\!]\ q & = & q \\ \mathcal{D}s[\![DecSeq(var, \mathrm{S})]\!]\ q & = & q[var/\mathcal{S}[\![\mathrm{S}]\!]] \\ \mathcal{D}s[\![DeclS(\mathrm{DS}_1, \mathrm{DS}_2)]\!]\ q & = & \mathcal{D}s[\![\mathrm{DS}_2]\!]\ (\mathcal{D}s[\![\mathrm{DS}_1]\!]q) \end{array}$$

4.2.7 Programme

Un programme spécifie d'une part les déclarations globales valables pour la recherche spécifique, et d'autre part la condition particulière que la sous-séquence recherchée doit vérifier.

Le sens du programme, donné par la fonction $\mathcal{P}$, consiste à faire la supposition qu'il existe un environnement de variables possédant le même domaine de définition que l'environnement de variables obtenu par les déclarations du programme, et à dire que le programme renverra la valeur *true* s'il *existe* une sous-séquence dans la séquence fondamentale qui vérifie la condition donnée.

Insistons sur le fait que cette sémantique est bien déclarative : on dit dans quelles conditions on peut trouver la sous-séquence recherchée, à savoir s'il *existe* un environnement de variables et une sous-séquence vérifiant la condition spécifiée, mais en aucun cas on ne précise la façon dont cette sous-séquence va être trouvée dans la séquence fondamentale.

Rappel de la syntaxe abstraite

$\mathrm{P} = prog(\mathrm{DV}, \mathrm{DS}, \mathrm{S})$

Signature de la fonction

$\mathcal{P}$: Program $\rightarrow$ $\mathbb{S}$equence $\rightarrow$ $\mathbb{B}$ool

Définition de la fonction

$$\begin{array}{rcl} \mathcal{P}[\![prog(\mathrm{DV}, \mathrm{DS}, \mathrm{F})]\!]\ s & = & \exists ss \in \mathbb{E}\mathrm{vent}^+ \\ & \underline{et} & \exists v \in \mathbb{E}\mathrm{nv}\mathbb{V}\mathrm{ar} \\ & \underline{et} & domdef(v) = domdef(\mathcal{D}v[\![\mathrm{DV}]\!]v_0) \\ & \underline{et} & q = \mathcal{D}s[\![\mathrm{DS}]\!]\ q_0 \\ & \underline{et} & \mathcal{S}[\![\mathrm{F}]\!]\ v\ s\ ss\ q = true \\ & \underline{avec} & v_0 = \_ \rightarrow undef \\ & \underline{et} & q_0 = \_ \rightarrow not\_def \end{array}$$

La fonction *domdef* utilisée ci-dessus prend pour argument une fonction et renvoie son domaine de définition :

Soit $f : A \rightarrow B$

Alors $\forall\, a \in A\ :\ a \in\ domdef(f) \iff f(a) \neq undef$

Chapitre 5

Implémentation du langage LaDAA

Nous explicitons dans ce chapitre les différentes étapes parcourues pour passer de la définition théorique du langage LaDAA à son implémentation.

5.1 Introduction

Nous avons choisi, plutot que de réutiliser le code d'ASAX afin de construire un compilateur du langage LaDAA, de réaliser un *traducteur* du langage LaDAA en langage RUSSEL. Le code RUSSEL ainsi généré doit donc ensuite être exécuté avec ASAX. Ce choix constituait, selon nous et au vu des échéances fixées, la solution la plus réalisable dans le cadre de ce mémoire.

La traduction du langage LaDAA en langage RUSSEL ne s'avère malheureusement pas directe, tout particulièrement pour la traduction des conditions élaborées de type ensemble.

C'est pourquoi nous avons opté pour une solution intermédiaire consistant à utiliser les automates finis non déterministes dans la traduction du langage LaDAA en langage RUSSEL.

Dès lors, nous expliquons, dans un premier temps, comment on peut transformer un programme LaDAA en un automate qui lui correspond. Dans un deuxième temps, nous voyons comment transformer cet automate en langage RUSSEL. Ainsi, le code RUSSEL finalement généré devra réaliser la requête initiale écrite en LaDAA. Remarquons que, comme il est expliqué dans [2], la transformation d'automates en langage RUSSEL peut se faire de façon directe. Ensuite, nous exposons l'implémentation du traducteur réalisée en langage C et donnons les limites de celle-ci. Nous finissons par donner le code RUSSEL généré pour l'exemple des échecs de login évoqué plus haut (voir chapitre 2).

5.2 Transformation du langage LaDAA en automates

5.2.1 Définition des automates

Définition intuitive

Définissons de façon intuitive la notion d'automate en tant que graphe. Un automate est constitué d'une ensemble de noeuds et d'un ensemble d'arcs. Notons que dans tous les automates que nous avons considérés, ces deux ensembles sont toujours non vides.

Ainsi, tout automate possède un nombre fini quelconque et non nul de noeuds. En outre, tout automate possède un et un seul noeud initial, ainsi qu' un et un seul noeud final (restriction au cas qui nous intéresse). Nous utilisons les représentations graphiques de la figure 5.1 pour désigner les noeuds.

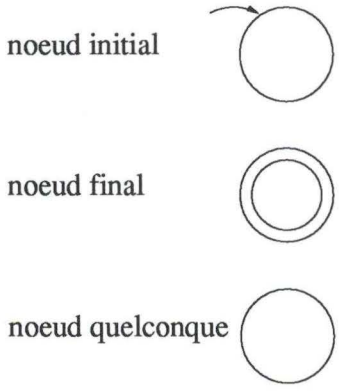

Figure 5.1: Représentation graphique des différents noeuds d'automate

De plus, tout automate possède un nombre fini quelconque et non nul d'arcs allant d'un noeud vers un autre. Tout arc est monté d'une condition correspondant, dans le cas des automates considérés ici, à une condition simple du langage LaDAA. Tout noeud non final de l'automate est origine d'un ou plusieurs arcs. Le noeud final n'est origine d'aucun arc. Un exemple d'automate est donné à la figure 5.2. Remarquons que tous les automates considérés dans ce mémoire correspondent à des graphes acycliques.

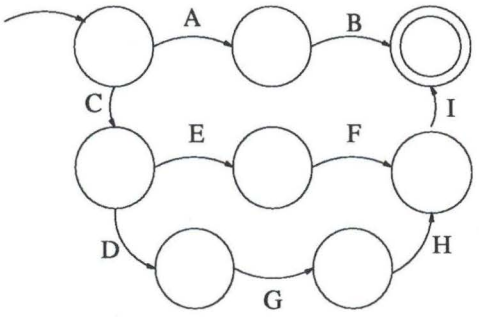

Figure 5.2: Exemple d'automate

Définition mathématique

On peut définir[1] un automate fini M comme un 5-tuplet $(S, s_0, A, \Sigma, \delta)$ où :

- S est un ensemble fini non vide d'états,
- $s_0 \in S$ est l'état initial,
- $A \subseteq S$ est l'ensemble des états acceptants[2] (dans notre cas, il est réduit à un singleton et sera donc noté a),
- Σ est un alphabet (il est dans notre cas infini et composé des conditions simples du langage LaDAA),
- δ est une relation de $S \times \Sigma$ dans S qui est définie partout sur $S \times \Sigma$, à savoir que pour chaque paire $(s, C) \in S \times \Sigma$, il existe au moins un état $t \in S$ qui est tel que $t = \delta(s, C)$. On appelle δ la relation de transition de l'automate M.

Un automate fini définit un graphe de transition dont les noeuds sont les états de l'automate, et où il y a un arc allant du noeud s au noeud t et labellé par C si $t = \delta(s, C)$. Par abus de langage, nous parlerons aussi bien de noeuds que d'états pour désigner les états d'un automate.

Un automate fini est dit *déterministe* si δ est une fonction, à savoir que toutes les transitions partant d'un état donné ont un label distinct. Un automate *non déterministe* peut contenir un état avec plusieurs arcs distincts partant de cet état et portant un label identique. Dans ce cas, nous noterons $t \in \delta(s, C)$ au lieu de $t = \delta(s, C)$. Tous les automates considérés dans notre exposé sont non déterministes.

5.2.2 Exécution des automates : utilisation des automates pour la recherche

Une fois traduit en automate, l'exécution d'un programme LaDAA correspond au parcours de l'automate qui lui correspond. Voyons donc comment s'exécute un automate et comment il va effectuer la recherche qui lui correspond. On peut donner deux versions équivalentes d'exécution : non déterministe et complète. L'implémentation en ASAX devrait réaliser la version complète.

Version non déterministe

On a une suite d'états ϵ composés :

1. d'un suffixe de la suite d'événements à analyser que nous notons σ,
2. d'un environnement v des variables partiellement instanciées grâce à la partie déjà analysée et
3. d'un noeud courant.

La spécification intuitive d'une exécution de l'automate est que :

- soit elle répond *échec* : aucune sous-séquence n'est trouvée,
- soit elle répond *succès* : on a trouvé une valeur v qui justifie l'existence d'une sous-séquence.

[1] Le lecteur curieux pourra consulter [5] (en particulier la lecture 3) pour plus d'informations au sujet des automates.

[2] Un état acceptant est aussi appelé état final.

Etat initial

Nous avons que :

1. σ est la suite complète d'événements à analyser,
2. v est l'environnement complètement indéterminé,
3. le noeud courant est le noeud initial.

Etat final

Nous avons que :

- soit σ est vide et le noeud courant n'est pas le noeud final,
- soit le noeud courant est le noeud final.

Transition

Nous avons $\sigma = e.\sigma'$ et le noeud courant n'est pas final. On choisit a priori une transition (un arc) issue du noeud courant (il en existe toujours une). Si la condition est vraie, l'état suivant est constitué de :

1. σ'
2. v' résultant de l'évaluation de la condition
3. le noeud cible de l'arc devient le noeud courant.

Si la condition est fausse, l'état suivant est constitué de :

1. σ'
2. v inchangé
3. le noeud courant est lui aussi inchangé.

On peut également réaliser ce dernier choix s'il existe une transition dont la condition est vraie.

Version complète

La version complète réalise en parallèle tous les choix possibles à partir d'un noeud. Il peut donc y avoir plusieurs noeuds courants. Dès lors, nous avons une suite d'états ϵ composés :

1. d'un suffixe de la suite d'événements à analyser que nous notons σ,
2. d'un ensemble $\mathcal{S}$ de couples $< v, n >$ où v est un environnement de variables partiellement instanciées grâce à la partie déjà analysée, et où n est un noeud courant,
3. d'un ensemble $\mathcal{C}$ de noeuds *collectés* : il s'agit de couples $< v, n >$ où n est le noeud final.

Ainsi, à chaque noeud courant correspond un et un seul environnement de variables. La spécification intuitive d'une exécution de l'automate est dans ce cas que :

- soit elle répond *échec* : aucune sous-séquence n'est trouvée,
- soit elle répond *succès* : on a trouvé un ensemble non vide de valeurs v qui justifient chacune l'existence d'une sous-séquence.

Etat initial

Nous avons que :

1. σ est la suite complète d'événements à analyser,
2. $\mathcal{S} = \{< v, s_0 >\}$ où v est l'environnement complètement indéterminé et s_0 est le noeud initial,
3. $\mathcal{C} = \emptyset$.

Etat final

Nous avons que σ est vide et :

- soit $\mathcal{C}$ est vide,
- soit $\mathcal{C}$ contient au moins un couple $< v, n >$.

Transition

Nous avons $\sigma = e.\sigma'$. Pour chacun des couples $< v, n >$ de $\mathcal{S}$, on peut effectuer toutes les transitions issues du noeud n où la condition est vraie. Remarquons bien que l'on peut ici aussi choisir de ne pas effectuer une transition dont la condition est vraie. Notons $c_1, c_2, \ldots, c_m$ les noeuds cibles des transitions effectuées. Dès lors, l'état ϵ devient :

1. σ'
2. $v_1, v_2, \ldots, v_m$ sont les environnements résultant de l'évaluation de la condition associée respectivement à la transition ayant pour cible $c_1, c_2, \ldots, c_m$, et $\mathcal{S} = \mathcal{S} \cup \{< v_1, c_1 >, < v_2, c_2 >, \ldots, < v_m, c_m >\} - \{< v, n >\}$
3. si parmi les noeuds $c_1, c_2, \ldots, c_m$ figure le noeud final (e.g. le noeud c_i), alors $\mathcal{C} = \mathcal{C} \cup < v_i, c_i >$; l'ensemble $\mathcal{C}$ est inchangé sinon.

5.2.3 Traduction des constructions du langage LaDAA en automate

Remarquons que dans toute cette partie, la terme *séquence* désigne la construction syntaxique représentant une condition élaborée de type séquence.

Le cas des conditions simples

Une condition simple est, en quelque sorte, l'élément atomique de toute condition, aussi élaborée soit elle. Elle correspond simplement à un automate du type de la figure 5.3 qui passe de l'état initial dans l'état final dès que la condition simple est rencontrée.

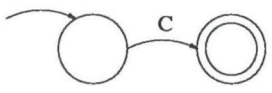

Figure 5.3: Automate d'une condition simple

Le cas des conditions élaborées de type séquence

Une séquence est une condition élaborée qui met à la suite plusieurs conditions simples, voire élaborées.

Construction intuitive de l'automate correspondant

Supposons dans un premier temps que nous avons une séquence de deux conditions simples. L'automate correspondant à cette séquence peut être obtenu à partir des automates des deux conditions en fusionnant le noeud final de l'automate de la première condition avec le noeud initial de l'automate de la seconde condition. La figure 5.4 illustre la fusion en séquence des automates de deux conditions simples.

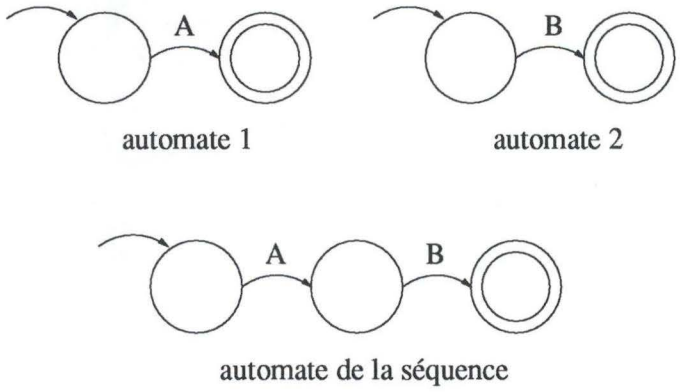

Figure 5.4: Automate d'une séquence de deux conditions simples

Remarquons que le même principe peut être appliqué à deux automates quelconques correspondant à une autre condition qu'une condition simple.

Supposons à présent que la séquence est composée de plus de deux conditions. Il suffit alors de fusionner en séquence les automates des deux premières conditions, puis de fusionner en séquence l'automate résultant de cette première fusion avec l'automate de la troisième condition. On répète ce processus jusqu'à ce que tous les automates des différentes conditions composant la séquence soient fusionnés. L'automate résultant correspond à l'automate de la séquence des conditions.

Construction mathématique de l'automate correspondant

Notons $M_1 = (S_1, s_{01}, a_1, \Sigma, \delta_1)$ et $M_2 = (S_2, s_{02}, a_2, \Sigma, \delta_2)$ deux automates définis sur un même alphabet. L'automate $M = (S, s_0, a, \Sigma, \delta)$ résultant de la fusion en séquence de M_1 *suivi de* M_2 est défini comme suit :

- $S = (S_1 \cup S_2) - a_1$,
- $s_0 = s_{01}$,
- $a = a_2$,
- $t \in \delta(s, C)$ si :

· soit $t \in \delta_1(s, C)$ et $s \in S_1$ (avec $t \neq a_1$)

· soit $t \in \delta_2(s, C)$ et $s \in S_2$

· soit $t = s_{02}$ et $a_1 \in \delta_1(s, C)$

Le cas des conditions élaborées de type union

Une union est une condition élaborée qui met en parallèle plusieurs conditions simples, voire élaborées.

Construction intuitive de l'automate correspondant

Procédons comme pour les séquences et supposons, dans un premier temps, que nous avons une union de deux conditions. A ces deux conditions correspondent deux automates. Pour obtenir l'automate de l'union de ces deux conditions, il suffit de fusionner le noeud initial du premier automate avec le noeud initial du second automate, ainsi que de fusionner le noeud final du premier automate avec le noeud final du second automate. La figure 5.5 illustre la fusion en union des automates de deux conditions simples.

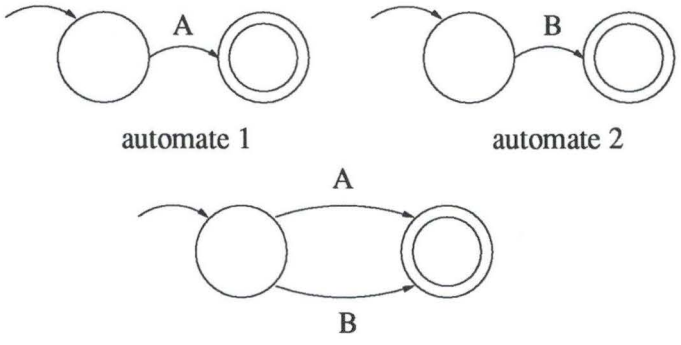

Figure 5.5: Automate d'une union de deux conditions simples

Si on suppose à présent que nous avons une union de plus de deux conditions, il suffit d'appliquer le même principe que celui utilisé pour les séquences, à savoir fusionner en union les automates des deux premières conditions, puis fusionner en union l'automate ainsi obtenu avec l'automate de la troisième condition. On continue ainsi jusqu'à ce que tous les automates des différentes conditions composant l'union soient fusionnés en union. L'automate résultant correspond à l'automate de l'union des conditions.

Construction mathématique de l'automate correspondant

Notons $M_1 = (S_1, s_{01}, a_1, \Sigma, \delta_1)$ et $M_2 = (S_2, s_{02}, a_2, \Sigma, \delta_2)$ deux automates définis sur un même alphabet. L'automate $M = (S, s_0, a, \Sigma, \delta)$ résultant de la fusion en union de M_1 et M_2 est défini comme suit :

- $S = (S_1 \cup S_2) - \{s_{02}, a_2\}$,

- $s_0 = s_{01}$,
- $a = a_1$,
- $t \in \delta(s, C)$ si :
  - soit $t \in \delta_1(s, C)$ et $s \in S_1$
  - soit $t \in \delta_2(s, C)$ et $s \in S_2$ avec $t \neq a_2$ et $s \neq s_{02}$
  - soit $t = a$ et $a_2 \in \delta_2(s, C)$
  - soit $s = s_0$ et $t \in \delta_2(s_{02}, C)$

Le cas des conditions élaborées de type ensemble

Ce cas est le plus délicat à traiter. La notion d'ensemble correspond ici à la notion mathématique d'un multi-ensemble : une collection non ordonnée d'éléments non nécessairement distincts.

Ensemble de deux conditions simples

Supposons dans un premier temps que nous avons un ensemble de deux conditions simples. Afin de clarifier les idées, nous numérotons les sommets des automates de sorte que chaque noeud est identifié par son numéro. La figure 5.6 nous montre les automates numérotés correspondant aux deux conditions simples. Intuitivement, spécifier une recherche de l'ensemble des conditions A et B signifie que l'on désire trouver soit la séquence AB, soit la séquence BA.

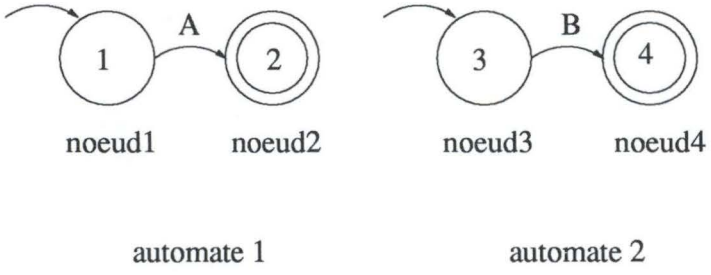

Figure 5.6: Automates numérotés des deux conditions simples

Pour obtenir l'automate correspondant à l'ensemble des conditions, nous introduisons aussi des automates numérotés avec deux indices : chaque numéro est alors un couple d'entier plutôt que simplement un entier. Par exemple, au lieu du noeud 1, nous avons le noeud $(1, 5)$. L'automate d'un ensemble de condition sera un automate doublement numéroté. Nous exposons à présent intuitivement comment construire cet automate.

Si nous avons les deux automates de la figure 5.6, il est raisonnable de dire que le noeud initial de l'automate final a pour indices les numéros des noeuds initiaux des deux automates de départ, et que le noeud final de l'automate a pour indices les numéros des noeuds finaux des deux automates de départ. Ainsi, pour notre exemple, le noeud initial de l'automate final est numéroté $(1, 3)$ et le noeud final est numéroté $(2, 4)$. Reste à construire le reste de l'automate.

A partir du noeud initial, nous considérons le premier indice et regardons dans le premier automate s'il existe des arcs ayant pour origine le noeud portant cet indice. Pour chacun de ces arcs, nous créons un nouveau noeud dans l'automate final avec pour premier indice le numéro du noeud cible de l'arc, et pour second indice le second indice du noeud initial. Nous traçons ensuite un arc du noeud initial vers ce nouveau noeud. Pour notre exemple, il s'agit de créer le noeud $(2,3)$ et de tracer l'arc allant du noeud $(1,3)$ vers le noeud $(2,3)$.

Nous procédons de façon analogue pour le second indice du noeud initial en considérant cette fois les arcs du second automate. Nous créons alors, toujours pour notre exemple, le noeud $(1,4)$ et l'arc joignant le noeud $(1,3)$ au noeud $(1,4)$.

Ce principe est ensuite appliqué à chacun des nouveaux noeuds créés. Notons que dans ce cas, il est possible qu'il ne faille pas créer de nouveau noeud (car il a peut-être déjà été créé par ailleurs) mais simplement un arc. C'est le cas dans notre exemple pour les noeuds $(2,3)$ et $(1,4)$ qui sont reliés au noeud final $(2,4)$. La figure 5.7 montre l'automate final de l'exemple de la figure 5.6.

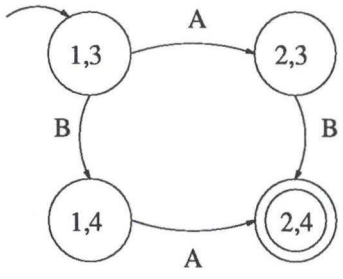

Figure 5.7: Automate doublement numéroté de l'ensemble des deux conditions simples

Il reste ensuite à transformer cet automate doublement numéroté en automate simplement numéroté. Remarquons que les noeuds peuvent être numérotés de façon tout à fait indifférente, pourvu que chaque noeud puisse être identifié par son numéro. La figure 5.8 indique une renumérotation possible de l'automate de la figure 5.7.

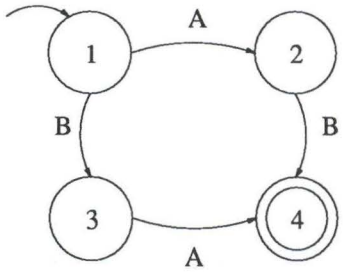

Figure 5.8: Automate de l'ensemble des deux conditions simples

Ensemble de trois conditions simples

Nous supposons à présent que nous avons un ensemble de trois conditions simples. Intuitivement, spécifier une recherche de l'ensemble des conditions A, B et C signifie que l'on désire trouver une des séquences ABC, ACB, BAC, BCA, CAB ou CBA.

Pour ce faire, il suffit de construire l'automate de l'ensemble des deux premières conditions A et B comme nous venons de l'exposer plus haut, puis de réappliquer l'idée générale d'automate doublement numéroté ainsi que celle de création de nouveaux noeuds et de nouveaux arcs à partir de l'automate ainsi créé et de l'automate de la condition simple C. La figure 5.9 nous montre la construction progressive de l'automate de l'ensemble des trois conditions A, B et C.

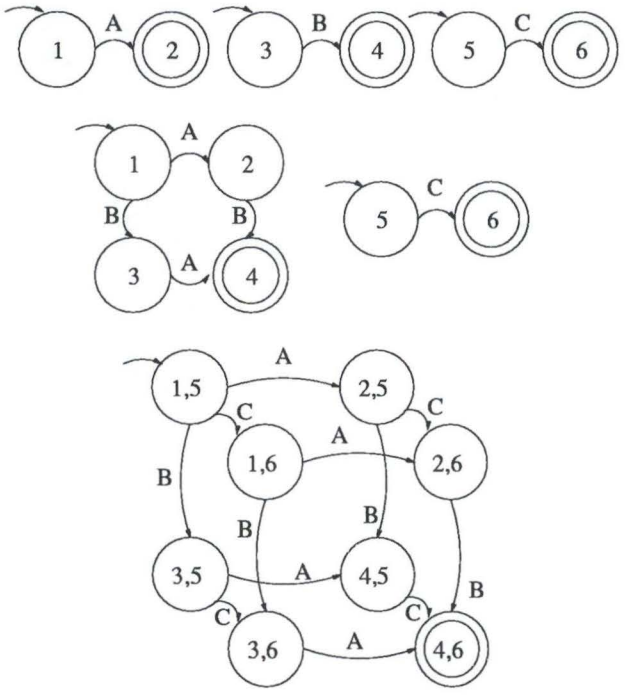

Figure 5.9: Automate de l'ensemble des trois conditions simples

Pour les cas de plus de trois conditions, on applique récursivement ce même principe de fusion progressive des automates des conditions jusqu'à n'obtenir plus qu'un seul automate. Cet automate est l'automate de l'ensemble des conditions.

Algorithme général

Ayant fixé notre intuition sur des exemples simples, nous pouvons à présent donner une formulation générale du procédé de fusion ensembliste d'automates quelconques. Pour ce faire, nous exposons en pseudo-code l'algorithme général que nous avons implémenté. Nous supposons construits les deux automates de départ.

Le noeud initial de l'automate final a pour indices les numéros des noeuds initiaux des deux automates de départ, et le noeud final de l'automate final a pour indices les numéros des noeuds finaux des deux automates de départ.

La boucle principale de l'algorithme est articulée autour de la variable `aparcourir` qui est une liste contenant l'ensemble des sommets que l'algorithme doit encore "parcourir". La variable `courant` représente le premier élément de cette liste.

On considère chaque noeud l'un après l'autre: chaque noeud de la liste `aparcourir` devient le noeud `courant` à son tour. Pour le noeud courant, on regarde s'il existe un (ou plusieurs) arc(s) dans le premier automate dont l'origine corresponde au premier indice du noeud courant. Pour chacun de ces arcs, on regarde si le noeud, indicé par le numéro cible de l'arc et la seconde coordonnée du noeud courant, existe déjà dans l'automate. Si ce n'est pas le cas, on le crée et on l'ajoute à la liste des noeuds à parcourir. On crée un arc entre le noeud courant et ce noeud que l'on vient de créer (ou qui était déjà présent dans l'automate). On procède de façon similaire avec les arcs du second automate.

Une fois le noeud courant examiné, on le retire de la liste `aparcourir` et le noeud suivant de la liste devient le noeud `courant`. Lorsqu'il n'y a plus de noeud à parcourir, on renumérote l'automate à double indice en automate à simple indice.

```
/* construction des noeuds initiaux et finaux */
 noeud_initial := (noeud_initial_autom1,noeud_initial_autom2);
 noeud_final := (noeud_final_autom1,noeud_final_autom2);
/* construction du reste de l'automate */
 arcs1 := liste_arcs_autom1;
 arcs2 := liste_arcs_autom2;
 aparcourir := {noeud_initial};
 courant := premier_element(aparcourir);
 tant que (aparcourir non vide) faire{
   pour chaque (arc) de arcs1 :
     si (origine(arc)=coord1(courant)) alors{
       si (not noeud_exist(cible(arc),coord2(courant))) alors{
          nouv_noeud := creer_noeud(cible(arc),coord2(courant));
          aparcourir := aparcourir + {nouveau_noeud};
       }
       creer_arc(courant,(cible(arc),coord2(courant)));
     }
   pour chaque (arc) de arcs2 :
     si (origine(arc)=coord2(courant)) alors{
       si (not noeud_exist(coord1(courant),cible(arc))) alors{
          nouv_noeud := creer_noeud(coord1(courant),cible(arc));
          aparcourir := aparcourir + {nouveau_noeud};
       }
       creer_arc(courant,(coord1(courant),cible(arc)));
     }
   aparcourir := aparcourir - {courant}
```

```
    courant := premier_element(aparcourir);
  }
  renumeroter_automate;
```

Construction mathématique de l'automate correspondant

Notons $M_1 = (S_1, s_{01}, a_1, \Sigma, \delta_1)$ et $M_2 = (S_2, s_{02}, a_2, \Sigma, \delta_2)$ deux automates définis sur un même alphabet. Il nous faut en outre définir un nouveau type d'automates où chaque état est un couple d'états. Nous appelons ce genre d'automate un *automate à double indice*. Notons un tel automate M.

L'automate $M = ((S_1 \times S_2), (s_{01}, s_{02}), (a_1, a_2), \Sigma, \delta)$ est l'automate de la fusion ensembliste des automates M_1 et M_2 si la relation de transition δ est telle que:

$(t, u) \in \delta((r, s), C)$ si

- soit $t = r$ et $u \in \delta_2(s, C)$
- soit $u = s$ et $t \in \delta_1(r, C)$

5.2.4 Problème des variables

Tout programme en langage LaDAA peut contenir des déclarations de variables. Chaque parcours de l'automate utilise une copie de l'ensemble des variables. Cette copie est transmise lors de chaque transition. Lorsque plusieurs transitions sont possibles, il faut créer autant de copies qu'il y a de transitions possibles.

Une optimisation consisterait à ne transmettre d'un noeud à l'autre que les variables qui sont encore nécessaires pour la suite (i.e. figurent encore dans une condition). Nous n'avons pas implémenté cette optimisation.

L'évaluation des conditions provoque une mise à jour de l'environnement des variables. Au départ, toutes les valeurs des variables sont indéfinies et le parcours de l'automate les détermine progressivement. Cette détermination progressive est l'implémentation du quantificateur existentiel que nous avons introduit dans la sémantique (voir 4.2.7). Le parcours de l'automate réalise une "preuve" de l'existence de l'environnement des variables v correspondant à ce quantificateur.

5.3 Transformation des automates en RUSSEL

La conversion d'un automate en code RUSSEL se fait de façon tout à fait directe. Nous distinguons ici les cas des noeuds initial et final de ceux des noeuds quelconques.

5.3.1 Le cas du noeud initial

Supposons que le noeud initial porte le numéro i. Le code RUSSEL à générer est le suivant :

```
init_action
 vars := undef ;
 trigger off for_next rule_i(vars)
end.
```

Notons que `vars` désigne l'ensemble des variables utilisées dans le programme. Rappelons que toutes ces variables sont de type *string*.

5.3.2 Le cas du noeud final

Supposons que le noeud final porte le numéro i. Le code RUSSEL à générer est le suivant :

```
rule rule_i(vars);
begin
 print(message);
 println(vars)
end ;
```

Le message en question est le message spécifié dans le programme LaDAA juste après le mot clé `performing print`.

5.3.3 Le cas d'un noeud quelconque

Version sans variables

Nous supposons ici que le langage LaDAA ne contient aucune variable. Nous supposons en outre que nous traitons le noeud 1 et que de ce noeud partent:

- un arc monté de `Cond_` 1 vers le noeud 2
- un arc monté de `Cond_` 2 vers le noeud 3
- un arc monté de `Cond_` 3 vers le noeud 4
- un arc monté de `Cond_` 4 vers le noeud 5

De plus, nous supposons que le noeud 5 est le noeud final de l'automate considéré. Le code RUSSEL à générer est le suivant :

```
rule rule_1(vars) ;
begin
 if Cond_1 --> trigger off for_next rule_2 fi ;
 if Cond_2 --> trigger off for_next rule_3 fi ;
 if Cond_3 --> trigger off for_next rule_4 fi ;
 if Cond_4 --> trigger off at_completion rule_5 fi ;
 trigger off for_next rule_1
end ;
```

Version avec variables

Supposons à présent que nous ayons des variables. La méthode la plus simple pour en tenir compte est de passer l'ensemble des variables à chacune des règles déclenchées. Ainsi, la construction de l'environnement des variables peut s'effectuer de manière progressive au cours de l'analyse de l'audit trail. Notons

que cette méthode n'est pas la plus économe mais c'est celle retenue dans le cadre de ce mémoire.

Le cas où l'on rencontre une condition du type `x=field` doit être traité de façon toute particulière: il faut tester si la variable est déjà définie ou non. Si celle-ci n'est pas encore définie, la condition devient une affectation du champ à la variable. Par contre, si la variable est déjà définie, la condition est une comparaison entre la valeur de la variable et la valeur du champ. Dès lors, le code suivant:

```
if x=field --> trigger off for_next rule_i(vars) fi ;
```

est remplacé par le code ci-dessous:

```
if x=undef  --> x := field fi ;
if x=field --> trigger off for_next rule_i(vars) fi ;
```

Au fur et à mesure de l'exécution de l'automate, les différentes variables, qui au départ étaient toutes indéfinies, vont prendre certaines valeurs. Ceci correspond à une construction progressive de l'environnement des variables que l'on supposait existant dans la partie sémantique. Remarquons que plusieurs environnements sont ainsi construits en parallèle pour chacun des parcours de l'automate (qui se fait de façon complète: on explore toutes les possibilités en parallèle). Il est donc possible qu'il existe plus d'un environnement de variables qui soit tel que la sous-séquence recherchée vérifie la condition donnée par le programme LaDAA.

Remarquons en outre que la condition `x=field` peut apparaître au sein d'une condition plus complexe du type `A and B and C`. Il convient dès lors de modifier le code suivant:

```
if A and B and C --> trigger off for_next rule_i(vars) fi ;
```

en:

```
eval_cond := 1 ;
if not (A) --> eval_cond := 0
fi;
if not (B) --> eval_cond := 0
fi;
if not (C) --> eval_cond := 0
fi;
if eval_cond=1 --> trigger off for_next rule_i(vars) fi;
```

Ceci correspond à évaluer la condition grâce à la variable `eval_cond` et à ne déclencher la règle que si cette variable vaut *true*. Ainsi, si une condition (`A`,`B` ou `C`) est du type `x=field`, il suffit de remplacer le `if-fi` qui lui correspond par le code donné plus haut afin de tenir compte des conditions de type `x=field`. Ce code doit cependant être quelque peu modifié pour respecter l'évaluation de `eval_cond`. Cela donne le code suivant:

```
eval_cond := 1 ;
if eval_cond=1 and x=undef  --> x := field
   eval_cond=1 and x!=undef --> if not(x=field)
                                   --> eval_cond := 0
                                fi
fi ;
if eval_cond=1 --> trigger off for_next rule_i(vars) fi;
```

Ainsi, l'évaluation de la condition `A and x=field and C` se traduira de la façon suivante :

```
eval_cond := 1 ;
if not (A) --> eval_cond := 0
fi;
if eval_cond=1 and x=undef  --> x := field
   eval_cond=1 and x!=undef --> if not(x=field)
                                   --> eval_cond := 0
                                fi
fi ;
if not (C) --> eval_cond := 0
fi;
if eval_cond=1 --> trigger off for_next rule_i(vars) fi;
```

Notons ici qu'un mécanisme semblable pourrait être mis en place pour les autres conditions impliquant une variable, comme par exemple une condition du genre `x < field`. Il suffirait de tester si la variable `x` est définie ou non, et d'arrêter l'analyse de l'audit trail dans le cas où elle ne l'est pas. Cependant, nous avons supposé, dans le cadre de ce mémoire, que ce type d'erreur de pouvait survenir et que toute variable, avant d'apparaître dans une comparaison de ce genre, était affectée d'une valeur de champ.

5.3.4 Ajout de is_active

En pratique, le nombre de règles déclenchées peut vite devenir assez important et rendre l'exécution d'un programme RUSSEL plutôt lourde à supporter par le système.

C'est pourquoi nous avons choisi d'introduire l'utilisation de la fonction prédéfinie `is_active`. Nous testons, avant tout déclenchement de règle, si celle-ci n'est pas déjà active, c'est-à-dire si elle n'a pas été déclenchée précédemment lors de l'analyse d'un enregistrement antérieur à l'enregistrement courant. On ne déclenche une règle que si celle-ci n'est pas encore active. Cette technique permet de limiter considérablement le nombre de règles actives, et ainsi d'alléger l'exécution ASAX.

L'intérêt de trouver plusieurs sous-séquences vérifiant une condition précise réside dans la valeur différente que les différentes recherches menées en parallèle ont attribué à l'environnement des variables lors de sa construction progressive. L'utilisation de la fonction `is_active` permet de ne trouver qu'une seule occurrence de chaque sous-séquence correspondant à un environnement de variables distinct. On évite dès lors de voir apparaître des "doublons" dans les sous-séquences trouvées et l'analyse de l'audit trail s'en trouve allégée.

Bien sûr, cette technique n'est plus efficace si on désire justement compter le nombre de sous-séquences qui vérifient une condition particulière, et devrait dès lors être modifiée. Nous nous sommes cependant limités à cette technique d'utilisation de la fonction `is_active` dans le cadre de ce mémoire.

5.4 Implémentation en langage C

5.4.1 Représentation interne du langage LaDAA

Il nous a paru judicieux de définir un représentation interne du langage LaDAA. Ainsi, par une simple lecture d'un programme en langage LaDAA, nous créons une représentation globale de ce programme. Un programme en langage LaDAA est principalement composé de deux parties: les déclarations de variables et de séquences prédéfinies d'une part, et la spécification de la condition que la séquence recherchée doit satisfaire d'autre part.

Déclarations de variables et de séquences prédéfinies

En ce qui concerne les variables, nous avons choisi de les stocker dans une liste simplement chaînée de chaînes de caractères. Nous avons dès lors défini les types de données suivants:

```
typedef struct vs *ListVars;
typedef struct vs{
   char nom[MAXCHAR];
   ListVars next;
}Vars;
```

Notons que la constante `MAXCHAR` désigne la taille maximale de toute chaîne de caractère utilisée dans notre implémentation. L'utilisation de cette constante permet une modification aisée de la taille maximale des chaînes de caractères. Remarquons, en outre, que l'utilisateur ne peut déclarer et utiliser que des variables de type *string*, puisque les seules affectations permises sont celles de noms de champs à des variables d'un programme LaDAA.

Le cas des séquences prédéfinies a été traité d'une façon similaire. Nous avons défini une structure de liste chaînée dont les éléments sont constitués, ici, de deux champs:

- le nom de la séquence prédéfinie qui correspond à l'identifiant donné dans la déclaration
- la définition de la séquence prédéfinie qui correspond au texte apparaissant dans la déclaration après le mot réservé `is`.

Les types de données que nous avons pour cela définis sont les suivants:

```
typedef struct sq *ListSequ;
typedef struct sq{
   char nom[MAXCHAR];
   char def[MAXCHAR];
   ListSequ next;
}Sequ;
```

Représentation interne des conditions

Il existe deux types de conditions: les conditions simples et les conditions élaborées telles que les séquences, les unions et les ensembles. Il était donc important de pouvoir faire cette distinction.

Nous avons dès lors choisi de représenter chaque condition (aussi bien simple qu'élaborée) par ce que nous appelons une boîte. Les différentes boîtes sont chaînées entre elles. Les types de données que nous avons définis sont les suivants :

```
typedef struct CondB *CondBoxPtr;
typedef struct CondB{
   int kind;
   union{
         char cond[MAXCHAR];
         CondBoxPtr condPtr;
   } elem;
   CondBoxPtr next;
}CondBox;
```

Les conditions simples sont représentées par une boîte contenant un entier valant 0 (indiquant ainsi que l'on a une condition simple),une chaîne de caractère contenant le texte de la condition et un pointeur vers la boîte suivante. Les conditions plus élaborées sont quant à elles représentées par une boîte contenant un entier valant 1 (indiquant ainsi que l'on a une condition élaborée), un pointeur vers une boite représentant cette condition et un pointeur vers la boîte suivante.

Un petit exemple parlant mieux qu'un long discours, considérons la condition suivante : `set_of(sequ_of(A followed_by B) coupled_with C)`.

La représentation interne de la figure 5.10 correspond à cette condition.

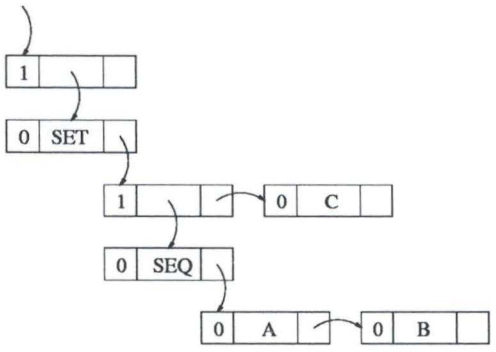

Figure 5.10: Représentation interne de l'exemple de condition

5.4.2 Limitations de l'implémentation

Deux caractéristiques du langage LaDAA n'ont pas été implémentées dans le cadre de ce mémoire. Premièrement, l'utilisation de la fonction *is_active* force le mode de recherche à être *first* par défaut. L'utilisateur n'a donc pas la possibilité de choisir ce mode. Deuxièmement, les restrictions sur le temps maximum séparant les événements de la sous-séquence recherchée ne sont pas impléméntées et la seule valeur possible est *norestriction*.

5.5 Exemple

Plusieurs exemples de programme LaDAA ont été implémentés et testés. Nous ne reprenons ici que le traditionnel exemple de recherche d'une séquence de trois échecs de login provenant d'un même utlisateur dont nous avons déjà parlé au chapitre 2.

En outre, le lecteur trouvera en annexe (voir annexe A) un second exemple qui est l'exemple présenté ici mais dans une version "ensembliste". Cet exemple illustre simplement l'allure complexe que peut prendre une traduction de programme LaDAA en langage RUSSEL. Il donne, bien entendu, les mêmes résultats que l'exemple présenté ici, à ceci près que son exécution sera plus lourde à supporter par le système. Le caractère déclaratif du langage LaDAA apparaît très clairement, en comparaison du style du langage RUSSEL.

Code LaDAA

```
with_declarations
var
string x
sequ toto is (strToInt(un_header_event)=6152 or
              strToInt(un_header_event)=6154 or
              strToInt(un_header_event)=6155)
          and strToInt(un_ret_error)!=0
          and x=un_subj_auid
find first sequ_of( toto followed_by toto followed_by toto )
with norestriction
performing print 'HELLO'
```

Automate

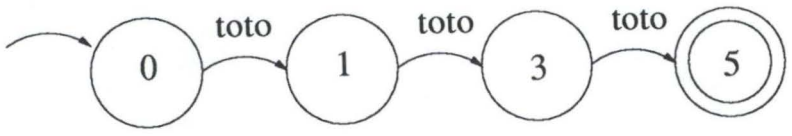

Figure 5.11: Automate correspondant à l'exemple

Code RUSSEL généré

```
rule rule_0(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
```

```
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid )
            --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_1(x_temp)
      --> trigger off for_next rule_1(x_temp)
   fi
fi;
trigger off for_next rule_0(x)
end;


rule rule_1(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid )
            --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_3(x_temp)
      --> trigger off for_next rule_3(x_temp)
   fi
```

```
fi;
trigger off for_next rule_1(x)
end;


rule rule_3(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp:= x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp:= un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid )
            --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active at_completion rule_5(x_temp)
      --> trigger off at_completion rule_5(x_temp)
   fi
fi;
trigger off for_next rule_3(x)
end;


rule rule_5(x:string);
begin
print(x);
println('HELLO')
end;


init_action;
var
x: string;
begin
```

```
x := '_undef_';
trigger off for_next rule_0(x)
end.
```

Résultats d'exécution

Le code ci-dessus a été exécuté avec ASAX sur un fichier d'audit de 8Mbytes en un temps se situant aux alentours de 30 secondes. L'exécution a bien fourni un message indiquant qu'une sous-séquence avait été trouvée. Remarquons que le temps d'exécution est fourni par ASAX à la fin de chaque exécution.

Conclusion et perspectives

Conclusion

Nous avons exposé ici le parcours suivi lors du développement du langage LaDAA, depuis la définition de sa syntaxe et de sa sémantique, jusqu'à son implémentation et sa traduction en langage RUSSEL. En particulier, nous avons exposé la transformation du langage LaDAA en automate, puis la traduction de ces derniers en langage RUSSEL.

Nous pourrions dire, pour utiliser un jargon populaire en informatique, que cette première version du langage LaDAA constitue en quelque sorte une version Béta. De nombreuses améliorations sont à apporter au langage LaDAA afin de l'étendre et l'optimiser.

Implémentation complète du langage

Tout d'abord, il conviendrait de trouver un moyen afin d'implémenter les caractéristiques du langage qui ne l'ont pas été, à savoir le *mode* de recherche et les *restrictions* sur la durée maximale que l'on peut imposer aux sous-séquences recherchées. Notons que d'autres types de restrictions pourraient également être imaginées et implémentées.

Recherche imbriquée

Ensuite, la recherche d'un scénario particulier pourrait s'effectuer en plusieurs étapes par raffinements successifs de la sous-séquence recherchée dans la séquence fondamentale :

1. recherche d'une sous-séquence dans une séquence fondamentale,
2. recherche d'une sous-séquence dans la sous-séquence trouvée au point 1,
3. recherche d'une sous-séquence dans la sous-séquence trouvée au point 2,
4. etc.

Chaque étape constitue ainsi en quelque sorte un filtre de plus en plus raffiné de la séquence fondamentale. Par conséquent, la syntaxe actuelle devrait être adaptée. On pourrait imaginer une syntaxe concrète modifiée de la façon suivante :

$$
\begin{array}{lcl}
< program > & ::= & \underline{with\_declarations} < declaration > \underline{find} < search > \\
< search > & ::= & < mode > < find > \\
 & | & < mode > < find > \underline{in} < search >
\end{array}
$$

et l'on pourrait par exemple avoir une recherche de sous-séquence qui ressemble à : `find first A in set_of(A coupled_with B coupled_with A)`.

Passage des paramètres

De plus, il serait intéressant d'optimiser le passage des paramètres dans les règles RUSSEL générées. Au lieu de passer systématiquement toutes les variables déclarées dans le programme LaDAA, une analyse du programme devrait pouvoir permettre de déterminer quelles variables sont encore utiles pour la suite et doivent dès lors encore figurer dans la liste des paramètres.

Etoile de Kleene

Remarquons en outre que le langage se basait en partie sur le concept des expressions régulières, exception faite de l'étoile de Kleene. On pourrait dès lors imaginer d'étendre le langage LaDAA en introduisant une construction implémentant une version modifiée de l'étoile de Kleene (avec limitation de la taille des sous-séquences).

Gestion des erreurs

En outre, un système de gestion des erreurs pourrait être développé. En effet, actuellement, aucune indication n'est fournie lorsqu'un programme LaDAA n'est pas écrit dans une syntaxe rigoureusement exacte, et un code RUSSEL tout à fait erroné peut parfois être généré. Dans tous les cas d'erreur, soit la traduction du langage LaDAA en RUSSEL provoque une erreur sans dire d'où elle vient exactement, soit l'exécution du code RUSSEL avec ASAX indique une erreur de syntaxe à une certaine ligne du code RUSSEL et donne un message d'erreur.

Preuves de correction

Finalement, une analyse du langage LaDAA pourrait être effectuée afin de prouver de manière formelle sa cohérence et déterminer sa complétude. En effet, tout comme en programmation logique, il est intéressant de se demander si le démonstrateur de théorème constitué par le parcours non déterministe de l'automate est cohérent et complet.

Bibliographie

[1] N. HABRA, B. LE CHARLIER, A. MOUNJI, and I. MATHIEU. ASAX : Software Architecture and Rule-based language for Universal audit trail analysis. In *Proceedings of the third European Symposium on Research in Security (ESORICS'92)*, Lecture Notes in Computer Science, Toulouse, November 1992. Springer-Verlag.

[2] A. MOUNJI. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* PhD thesis, Computer Science Institute, University of Namur, Belgium, September 1997.

[3] B. LE CHARLIER N. HABRA and A. MOUNJI. Advanced Security Audit trail analysis on Unix : Implementation Design of the NADF Evaluator. Research report 7/92, Computer Science Institute , University of Namur, March 1992.

[4] R.D. TENNENT. *Principles of Programming Languages.* Prentice-Hall, Englewood Cliffs (New Jersey), 1981.

[5] P. WOLPER. Francqui chair. Lectures given at the FUNDP, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, 1998. http://www.montefiore.ulg.ac.be/~pw/cours/francqui.html.

Annexe A

Exemple de programme LaDAA version ensembliste

Code LaDAA

```
with_declarations
var
string x
sequ toto is (strToInt(un_header_event)=6152 or
              strToInt(un_header_event)=6154 or
              strToInt(un_header_event)=6155)
          and strToInt(un_ret_error)!=0
          and x=un_subj_auid

find first set_of(toto coupled_with toto coupled_with toto)
with norestriction
performing print 'HELLO'
```

Automate

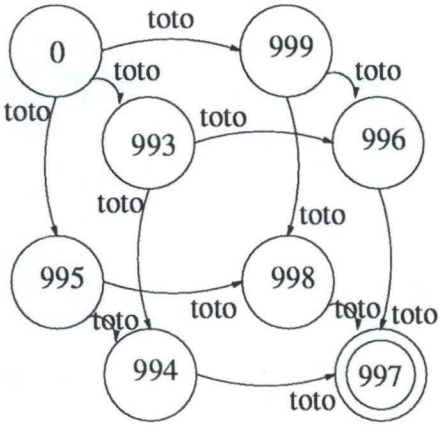

Figure A.1: Automate correspondant à la version ensembliste de l'exemple

Code RUSSEL généré

```
rule rule_0(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp:= x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp:= un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_999(x_temp)
      --> trigger off for_next rule_999(x_temp)
   fi
fi;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp:= un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_995(x_temp)
      --> trigger off for_next rule_995(x_temp)
   fi
```

```
fi;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_993(x_temp)
      --> trigger off for_next rule_993(x_temp)
   fi
fi;
trigger off for_next rule_0(x)
end;

rule rule_999(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_998(x_temp)
```

```
      --> trigger off for_next rule_998(x_temp)
   fi
fi;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_996(x_temp)
      --> trigger off for_next rule_996(x_temp)
   fi
fi;
trigger off for_next rule_999(x)
end;

rule rule_998(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
```

```
if eval_cond=1 -->
   if not is_active at_completion rule_997(x_temp)
      --> trigger off at_completion rule_997(x_temp)
   fi
fi;
trigger off for_next rule_998(x)
end;

rule rule_996(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active at_completion rule_997(x_temp)
      --> trigger off at_completion rule_997(x_temp)
   fi
fi;
trigger off for_next rule_996(x)
end;

rule rule_995(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
```

```
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_998(x_temp)
      --> trigger off for_next rule_998(x_temp)
   fi
fi;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_994(x_temp)
      --> trigger off for_next rule_994(x_temp)
   fi
fi;
trigger off for_next rule_995(x)
end;

rule rule_994(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
```

```
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active at_completion rule_997(x_temp)
      --> trigger off at_completion rule_997(x_temp)
   fi
fi;
trigger off for_next rule_994(x)
end;

rule rule_993(x:string);
var
eval_cond: integer;
x_temp: string;
begin
x_temp := x;
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_996(x_temp)
      --> trigger off for_next rule_996(x_temp)
   fi
fi;
```

```
eval_cond:=1;
if not (( strToInt ( un_header_event ) = 6152 or
          strToInt ( un_header_event ) = 6154 or
          strToInt ( un_header_event ) = 6155 ) )
      --> eval_cond:=0
fi;
if not (strToInt ( un_ret_error ) != 0 )
      --> eval_cond:=0
fi;
if (eval_cond=1 and x_temp='_undef_')
      -->  x_temp := un_subj_auid;
   (eval_cond=1 and x_temp!='_undef_')
      -->if not (x_temp = un_subj_auid ) --> eval_cond:=0
         fi
fi;
if eval_cond=1 -->
   if not is_active for_next rule_994(x_temp)
      --> trigger off for_next rule_994(x_temp)
   fi
fi;
trigger off for_next rule_993(x)
end;

rule rule_997(x:string);
begin
print(x);
println('HELLO')
end;

init_action;
var
x: string;
begin
x:= '_undef_';
trigger off for_next rule_0(x)
end.
```

Résultats d'exécution

Le code ci-dessus a été exécuté avec ASAX sur un fichier d'audit de 8Mbytes en un temps se situant aux alentours de 50 secondes. L'exécution a bien fourni un message indiquant qu'une sous-séquence avait été trouvée. Remarquons que le temps d'exécution est nettement supérieur au temps d'exécution du même exemple dans sa version "séquentielle".